XEROX

Xerox GlobalView

Document Interfaces Toolkit Software Reference

VP Series Applications

XEROX

Xerox GlobalView

Document Interfaces Toolkit Software Reference

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

Published August 1990.

Publication number: 610E22840

Printed in the United States of America

Changes are periodically made to this document. Changes, technical inaccuracies, and typographic errors will be corrected in subsequent editions.

This book was created using the Xerox 6085 Professional Computer System.

Table of Contents

5. Desktop Library 5-1

6. XString Library 6-1

7. Signals 7-1

8. XNS Library 8-1

1. Document IC Library

di_intro

NAME

di_intro - introductory explanation of document interchange functions

DESCRIPTION

The **DocIC** interface is a C-based programming tool that allows a person to create a new VP document or read an existing one. Also, new data may be added directly to the end of an existing VP document. The contents of an existing VP document may not be changed or deleted. But, through the use of an intermediary file, the contents of an existing VP file may be read up to a certain point and inserted within the intermediary file, the new data inserted, and the remainder of the VP document read. The same basic approach may be used to delete select data from a document: An existing VP document may be read up to a certain point and the information placed in an intermediary file. The undesired data may be skipped, and the remaining data is read and placed in the intermediary file.

The **DocIC** interface provides functions that may be used to create or read any of the basic VP document structures, such as text; fields; headings and footings; or frames of varying types.

Data is placed in a frame by the calling the **DocIC** interface functions that correspond to that particular type of frame. Currently, there are only two **IC** interfaces available that may be used to manipulate the contents of a frame. They are **GraphicsIC** and **TableIC**. **GraphicsIC** functions are used to create or read graphics frames and button frames. **TableIC** functions are used to create or read tables.

Document Creation

A VP document is initially created by calling either **di_start()** or **di_startap()**. Both of these two functions set up data structures for the document being created and return a handle to the newly created document. This handle is an identifier that is passed as an argument to other **DocIC** interchange functions as the means of identifying the document being manipulated.

The next step in creating a document is to add information to the document by calls to various **di_ap*()** functions. These functions are **di_apaframe()**, **di_apbreak()**, **di_apchar()**, **di_apfield()**, **di_apfntile()**, **di_apindex()**, **di_apnewpara()**, **di_appfc()**, **di_aptext()**, **di_aptofillin()**, **di_apfstyle()**, **di_appstyle()**, and **di_aptotxtlnk()**.

With regards to **di_apaframe()**, the function used to anchor a frame to an object in a document, the user typically calls various **GraphicsIC** or **TableIC** functions to create the contents of a frame, and then calls **di_apaframe()** to append that frame and its contents to the document. With regards to **di_starttext()**, the user calls **di_apaframe()** first and then calls **di_starttext()** to obtain a text handle. The handle returned by a call to **di_apaframe()** is then passed as an argument to **di_starttext()**.

di_apfield(), **di_apindex()** and **di_appfc()** all have return values. This allows the user to recursively call **di_ap*()** functions to add text and formatting information to fields, index, or PFC headers.

When all the desired data has been added to a document, call **di_finish()** to obtain a temporary reference, or handle. Then call the Desktop Library function **dsktp_move()** so that the resulting file may be placed on the VP desktop.

Document Enumeration

To enumerate the contents of an existing VP document, the first step is a call to **dsktp_getdocref()**. **dsktp_getdocref()** will return a handle for the specified document. Once the handle has been obtained, the contents may be manipulated. Next call **di_open()**. This function opens the specified document and returns **doc**, a handle for the document. Next, pass the handle and a **di_enumprocs** structure as arguments in a call to **di_enumerate()**. The **di_enumprocs** structure consists of a set of call-back procedures, where there is one call-back procedure for each of the corresponding object types that exist in the document. Objects, in this case, are defined as anchored frames, break characters, field, footnotes, indexes, new paragraphs, page format characters, or text.

The **di_enumerate()** function inspects a document from beginning to end. As different types of objects are encountered, this function calls the appropriate call-back procedure to process each particular type of object. Each call-back procedure returns a Boolean value. A value of **TRUE** terminates the enumeration. If **TRUE** is never returned, the enumeration continues to the end of the document.

Enumeration proceeds according to the "main flow" of text within a document. Main flow is considered to be the sequence of text that contains page format characters and frame anchor characters. This means that the call-back procedure, **di_aframeproc()**, will be called not when the frame itself is reached, but rather when the frame's anchor character is reached.

When the enumeration is complete, **di_close()** should be called to free all associated data structures and close any open file handles to the document.

Note that document creation and enumeration are totally separate activities and the functions and handles associated with one should not be used with the other. Enumeration is a read-only operation; no editing should be attempted while it is in progress. Likewise, enumeration should not be attempted when creating a document.

Data types

The basic data structure of the **DocIC** interface is **di_tcont** (text container). **di_tcont** may be defined as any object that can contain text. A **di_tcont** can be a caption, document, field, footing, heading, index, numbering, or text.

di_tcont is defined in DocIC.h as follows:

```
typedef struct {
    di_tcont_type type;
    union {
        di_caption      caption;
        di_doc          doc;
        di_field        field;
        di_footing      footing;
        di_heading      heading;
        di_index        index;
        di_numbering    numbering;
        di_text         text;
        } h;
    } di_tcont;
```

where, all elements inside the union **h** are unsigned integers.

di_tcont must contain at least one new paragraph character, since the paragraph properties of text are inherited from the preceding new paragraph character. The implementation of the **DocIC** interface automatically inserts the initial new paragraph characters as required. Therefore, it is always safe to assume they already exist. (You are free to append new paragraph characters, regardless. The implementation ensures that duplicate new paragraph characters do not appear in the document. The new paragraph characters inserted by the user have precedence over those inserted by the system.)

di_ins is a handle to specific instances of objects within a document. Many objects in a document may be uniquely identified and accessed via **di_ins**. In general, instances form the bridge between **DocIC** interfaces and the interfaces that are used specifically to manipulate the contents of frames, such as **GraphicsIC** and **TableIC**: **DocIC** interfaces provide an instance which may be passed to other Interchange interfaces. No object in any document may be accessed via **di_insnil**.

Table of DocIC Interfaces

The following table summarizes **DocIC** interfaces.

Object	Creating		Reading	
	Function	Page	Function	Page
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			
Anchored Footnote	di_setfnprops		di_aframeproc	
	di_apaframe		di_fnpropsproc	
	di_fntile		di_getfnprops	
	di_relcap		di_fntileproc	
Other Anchored Frame	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			
Newpara	di_apnewpara		di_newparaproc	
	di_setpara			

Object	Creating		Reading	
	Function	Page	Function	Page
Page (Footing/Heading/ Numbering)	di_appfc		di_pfcproc	
	di_relhead		di_docproc	
	di_relfoot			
	di_relnum			
Soft Page Break			di_sfbrkproc	
Fill-In Order	di_aptofillin		di_fillinproc	
	di_clearfillin		di_enumfillin	
Style	di_start		di_enumstyle	
	di_styleproc		di_fstyleproc	
	di_apfstyleproc		di_pstyleproc	
	di_appstyleproc			
	di_apfstyle			
	di_appstyle			
Text Link	di_aptotxtlnk		di_txtlnkproc	
	di_cleartxtlnk		di_enumtxtlnk	
Mode	di_setmode		di_getmode	

di_abort

NAME

di_abort - abort document creation

SYNOPSIS

```
#include "DocIC.h"

int
di_abort(doc)
    di_doc *doc;
```

DESCRIPTION

The **di_abort()** function is used to terminate the document generation process and deallocate the storage resources allocated to the document being terminated. This function's one argument is **di_doc**, the file handle returned by an earlier call to **di_start()** or **di_startap()**.

RETURN VALUE

If the call is successful, 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_abort() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

di_apaframe

NAME

di_apaframe - append anchored frame

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apaframe(to, type, frame, cont, wtcap, wbcap, wlcap, wrcap, font, trustsize, ret)
    di_tcont *to;
    di_aframetype type;
    dp_frameprops *frame;
    di_ins cont;                    /* di_insnil */
    dp_bool wtcap;                  /* FALSE */
    dp_bool wbcap;                  /* FALSE */
    dp_bool wlcap;                  /* FALSE */
    dp_bool wrcap;                  /* FALSE */
    dp_fontprops *font;             /* NULL */
    dp_bool trustsize;              /* FALSE */
    ret_apaframe *ret;              /* Returned */
```

DESCRIPTION

The **di_apaframe()** function is used to append an anchored frame to the text container specified by **di_tcont**. The resulting frame will be of a specific type and it will have specific format properties.

to is a pointer of the type **di_tcont**. It is a structure that defines the type of object contained within it and a handle to the object itself. **di_tcont** consists of a union of two members, **type** and **h**. The object type is defined by the member **type**. **type** is of the type **di_tcont_type**. It is an enumerated variable that may be set to one of the following values:

TC_CAPTION
TC_DOC
TC_FIELD
TC_FOOTING
TC_HEADING
TC_INDEX
TC_NUMBERING
TC_TEXT

The **h** member of **di_tcont** is an opaque variable that is to contain a handle returned by a previous call to a related handle generating function. It may contain one the following types:

di_caption
di_doc
di_field
di_footing
di_heading
di_index
di_numbering
di_text

The user specifies the handle type and its contents. In the case of **di_apaframe()**, the type is to be set to **TC_DOC** and the handle is to contain the return value of either **di_start()** or **di_startap()**. Appending an anchored frame to a caption, text, heading, footing, or numbering container is not allowed.

The **type** argument is of the type **di_aframetype**. It is an enumerated variable that specifies the type of anchored frame to be appended to the document container. It may be set to one of the following values:

AF_CUSP	Cusp Button
AF_GRAPH	Graphics
AF_TABLE	Table
AF_TEXT	Text
AF_FNOTE	Footnote
AF_OTHER	Other type

The **frame** argument is a pointer of the type **dp_frameprops**, a structure containing variables that control the appearance, dimensions, and page numbering of the frame in question.

The **cont** argument is the contents to be inserted in the frame. Currently, only interfaces that support the creation of graphic, table, text, and button frames are available.

The **w*cap** argument specifies the captions the frame should have.

font specifies the font properties of the frame anchor. Changing the font properties of the anchor does not affect the appearance of the anchor, but it does affect the default properties that succeeding characters will inherit.

trustsize is a Boolean value that controls the dimensions of the frame. If **trustsize** is set to **TRUE**, the frame size specified in **frame** will be used without modification. If set to **FALSE**, the frame size specified in **frame** will be ignored and the frame will be adjusted to fit the existing frame. This argument may only be set to **TRUE** when manipulating anchored table frames.

The return information is set into the structure **ret_apaframe**. It contains the following members:

di_ins frame;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;

The return information contains handles to the frame and its captions. The caption handles will be non-**NULL** only if the user specifies **TRUE** for the corresponding **w*cap** parameter. The user must later release each valid caption handle with calls to **di_relcap()**.

frame is a pointer of type **ret_apaframe**. The handle contained in **ret_apaframe** is passed as an argument in calls to **di_starttext()** and **gi_setgframeprops()**. It is not mandatory to call **di_starttext()** after calling **di_apaframe()**. Failure to call **di_starttext()** will only result in an empty text frame, except for the presence of one new paragraph character that has default paragraph and font properties.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apaframe() will fail if one or more of the following is true:

Doc_ContainerFull	No more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

SEE ALSO

di_relcap(), di_starttext(), gi_setgframeprops()

di_apbreak

NAME

di_apbreak - append break character

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apbreak(to, brprops, foprops)
   di_tcont *to;
   dp_breakprops *brprops;
   dp_fontprops *foprops;                    /* NULL */
```

DESCRIPTION

The **di_apbreak()** function is used to append a page break character to the container specified by **di_tcont**.

Refer to **di_apaframe()** for a description of **di_tcont**. Note that heading, footing and numbering containers may not be used.

brprops are the properties of the break character. Refer to the *DocICProps* section of this manual and the VP reference manuals for more information regarding text frame properties.

foprops are the font properties of the break character. The addition of these properties will not affect the appearance of the character itself, but will affect the properties that succeeding characters will inherit.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apbreak() will fail if one or more of the following is true:

Doc_ContainerFull	There is no more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

di__apchar

NAME

di__apchar - append character

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "XString.h"

int
di_apchar(to, c, foprops, num)
   di_tcont *to;
   XChar c;
   dp_fontprops *foprops;          /* NULL */
   unsigned num;                   /* 1 */
```

DESCRIPTION

The **di_apchar()** function is used to append one or more instances of the text character **c** to the specified **di_tcont**. Refer to **di_apaframe()** for a description of **di_tcont**.

The **num** argument specifies the number of times the character specified in **c** will be appended to the text container. The **foprops** argument specifies the font properties of the character(s).

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apchar() will fail if one or more of the following is true:

Doc_ContainerFull	There is no more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

di_apfield

NAME

di_apfield - append field

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apfield(to, fiprops, foprops, ret)
    di_tcont *to;
    dp_fldprops *fiprops;
    dp_fontprops *foprops;          /* NULL */
    di_field *ret;                  /* Returned */
```

DESCRIPTION

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

Refer to **di_apaframe()** for a description of **di_tcont**. Note that a field may not be appended to a heading, footing or numbering container.

di_apfield() returns a handle of type **di_field**. This handle is passed as an argument to other **di_ap*()** functions in order to add data to the newly appended field. It cannot be specified as the **di_tcont** in another call to **di_apfield()**. After appending data to a field, the field must be released by a call to **di_relfield()**.

The **fiprops** and **foprops** arguments specify field and font properties, respectively. Refer to the *dp_*props* section of this manual and the VP reference manuals for more information regarding font and field properties.

The fill-in order of a fields cannot be set when they are appended to a document. To specify the fill-in order of fields, use the **di_aptofillin()** function.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apfield() will fail if one or more of the following is true:

Doc_ContainerFull	There is no more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

SEE ALSO

di_relfield(), di_aptofillin()

di_apfntile

NAME

di_apfntile - append footnote reference tile

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apfntile(to, foprops)
    di_text to;
    dp_fontprops *foprops;            /* NULL */
```

DESCRIPTION

The **di_apfntile()** function is used to append a Footnote Reference Tile to the text container specified in the **di_text**. argument.

The **foprops** argument specifies the font properties of the newly generated Footnote Reference Tile.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apfntile() will fail if one or more of the following is true:

Doc_ContainerFull	There is no more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

di_apfstyle, di_appstyle

NAME

di_apfstyle, di_appstyle - append font and paragraph style

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apfstyle(doc, props)
    di_doc doc;
    dp_fstyleprops *props;

int
di_appstyle(doc, props)
    di_doc doc;
    dp_pstyleprops *props;
```

DESCRIPTION

The **di_apfstyle()** and **di_appstyle()** functions are used to append respective font and paragraph style properties to the styles in a document. Refer to the *Document Editor: Basics User Guide* for more information on document styles.

There are two ways to append styles. The first way is via the **styledat** argument to **di_start()**. It is used to define the style of first the new paragraph and page format characters. The second way is via calls to **di_apfstyle()** and **di_appstyle()**. These two functions are used to define subsequent style definitions. **di_apfstyle()** and **di_appstyle()** cannot be used to set the style of the first new paragraph and page format characters.

The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

The **props** argument is a pointer of the type **dp_fstyleprops** or **dp_pstyleprops**. It specifies the properties desired by the user.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_appstyle() and **di_apfstyle()** will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumstyle()

di_apindex

NAME

di_apindex - append index character

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apindex(to, ixprops, foprops, ret)
   di_tcont *to;
   dp_indexprops *ixprops;
   dp_fontprops *foprops;          /* NULL */
   di_index *ret;                  /* Returned */
```

DESCRIPTION

The **di_apindex()** function is used to append an index character to the text container specified in **di_tcont**.

Refer to **di_apaframe()** for a description of **di_tcont**. Note that heading, footing and numbering containers may not be specified.

The **ixprops** and **foprops** arguments specify the respective index and font properties to be assigned to the index.

di_apindex() returns **di_index**, a handle that may be used by other **di_ap*()** calls to add data to the index character. The **di_index** handle must be released via **relindex()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apindex() will fail if one or more of the following is true:

Doc_ContainerFull	There is no more room to append to this container.
Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_ObjIllegalInCont	Attempted to add an object of an unsupported type to a container.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
Doc_Unimpl	This function is not supported.

SEE ALSO

relindex()

di_apnewpara

NAME

di_apnewpara - append new paragraph characters

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_apnewpara(to,prprops,foprops,num)
    di_tcont *to;
    dp_paraprops *prprops;          /* NULL */
    dp_fontprops *foprops;          /* NULL */
    unsigned num;                   /* 1 */
```

DESCRIPTION

The **di_apnewpara()** function is used to append one or more new paragraph characters to the text container specified in the **di_tcont** argument. Refer to **di_apaframe()** for a description of **di_tcont.**

The **prprops** and **foprops** arguments specify the respective paragraph and font properties of the new paragraph. If **prprops** is **NULL**, the new paragraph inherits the props of the previous paragraph. If **foprops** is **NULL**, the new paragraph inherits the paragraph properties of the previous paragraph.

The **num** argument is a cardinal number that indicates the number of paragraph characters to be appended.

The **di_tcont** argument must contain at least one new paragraph character. The current implementation of this C interface automatically supplies the initial new paragraph character to the beginning of a new document. Additional new paragraph characters may be added. If the user adds a new paragraph character to the beginning of the document, only the user-supplied new paragraph character will be present.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_apnewpara() will fail if one or more of the following is true:

Doc_ContainerFull There is no more room to append to this container.

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_ObjIllegalInCont Attempted to add an object of an unsupported type to a container.

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

Doc_Unimpl This function is not supported.

di_appfc

NAME

di_appfc - append page format character

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_appfc(to, pgprops, foprops, whead, wfoot, wnum, ret)
   di_tcont *to;
   dp_pageprops *pgprops;
   dp_fontprops *foprops;            /* NULL */
   dp_bool whead;                    /* FALSE */
   dp_bool wfoot;                    /* FALSE */
   dp_bool wnum;                     /* FALSE */
   ret_appfc *ret;                   /* Returned */
```

DESCRIPTION

The **di_appfc()** function is used to append a page format character to the text container specified in the **di_tcont** argument. Only document, field and index containers may be used. Refer to **di_apaframe()** for a description of **di_tcont**.

The **pgprops** argument specifies the format characteristics of the resulting page character. When specifying page margin properties for the **pgprops** argument, the margins must be set so that at least one inch is available for text. An inch is equivalent to 72 points. For example, (left margin + right margin + 72 < = page width), and (top margin + bottom margin + 72 < = page height).

The **foprops** argument specifies the font properties of the page format character.

The **whead, wfoot** and **wnum** arguments are Boolean variables that are used to specify whether or not the resulting page format character will contain heading, footing , and/or numbering properties.

di_appfc() returns **ret_appfc,** a structure containing the following members:

```
di_heading lhead;
di_heading rhead;
di_footing lfoot;
di_footing rfoot;
di_numbering num;
```

The heading, footing and/or numbering handles will be NULL unless the user sets **whead, wfoot** and/or **wnum** to TRUE.

If the heading, footing and/or numbering handles are valid, the user can then apply them as text containers in calls to other **di_ap*()** functions. If the headers are to be the same on both left and right pages, only **lhead** should contain the heading. **rhead** should be left NULL. The same rule applies to **lfoot** and **rfoot**.

When specifying heading, footing or numbering, note that there are no automatic positioning parameters for information in headers and footers. The user must call the appropriate **di_ap*()** function to add the desired text and to position it with standard text formatting, such as spaces, paragraph alignment, leading, line height, and tabs.

Page number patterns are not recognized. To specify a page number in heading, footing, or numbering format parameters, insert a special character at the location in which a page number is desired. Note that the function **dp_getpagedel()** returns this special character.

When finished with heading, footing, and/or numbering parameters, every non-**NULL** parameter must be terminated by a call to **di_relhead()** , **di_relfoot()** or **di_relnum()**, respectively.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_appfc() will fail if one or more of the following is true:

Doc_ContainerFull There is no more room to append to this container.

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_ObjIllegalInCont Attempted to add an object of an unsupported type to a container.

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

Doc_Unimpl This function is not supported.

SEE ALSO

dp_getpagedel(), di_relhead() , di_relfoot(), di_relnum()

di_aptext

NAME

di_aptext - append text

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "XString.h"

int
di_aptext(to, text, foprops)
   di_tcont *to;
   XString text;
   dp_fontprops *foprops;        /* NULL */
```

DESCRIPTION

The **di_aptext()** function is used to append the text string specified in the **text** argument to the text container specified in the **di_tcont** argument. Refer to **di_apaframe()** for a description of **di_tcont**.

The resulting text will have the font properties specified in the **foprops** argument. If **foprops** is left **NULL** then text will inherit the font properties of the previous paragraph.

The **text** argument may not contain new paragraph characters (i.e., [set: 0, code: 35B]).

Use the **di_apnewpara()** function to append new paragraph characters.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_aptext() will fail if one or more of the following is true:

Doc_ContainerFull There is no more room to append to this container.

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_ObjIllegalInCont Attempted to add an object of an unsupported type to a container.

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

Doc_Unimpl This function is not supported.

SEE ALSO

di_apnewpara()

di_aptofillin

NAME

di_aptofillin - append item to fill-in order

SYNOPSIS

```
#include "DocIC.h"
#include "XString.h"

int
di_aptofillin(doc, name, type)
    di_doc doc;
    XString name;
    di_fillintype type;
```

DESCRIPTION

The **di_aptofillin()** function is used to append to the fill-in order of fields and tables. Refer to the *Document Editor: Basics User Guide* for more information on fill-in orders of fields and tables. The fill-in order of fields cannot be set once they have been appended to a document, except by calling **di_aptofillin()**.

The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**. It contains the field or table in question. The **name** argument identifies the object to be added to the fill-in order. The **type** argument specifies the type of object to be added to the fill-in order. The value of **type** may be one of the following:

```
FI_FIELD        /* field */
FI_TABLE        /* table */
```

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_aptofillin() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumfillin(), di_clearfillin()

di_aptotxtlnk

NAME

di_aptotxtlnk - append item to text link

SYNOPSIS

```
#include "DocIC.h"

int
di_aptotxtlnk(doc, item)
   di_doc doc;
   di_textlink *item;
```

DESCRIPTION

The **di_aptotxtlnk()** function is used to append an item to the end of the text frame link order. It may be either an existing text frame link order or one that had been cleared via **di_cleartxtlnk()**. Refer to the *Document Editor: Basics User Guide* for information on text frame link order.

The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**. It must contain the text frame handle and may, optionally, contain the text frame link order

The **item** argument is a pointer of the type **di_textlink**. It specifies a structure whose members define the item to be appended and the text format parameters to be assigned that item. It contains the following members:

```
XString name;
int partab;
dp_bool newpara;
dp_bool newline;
dp_bool paratab;
```

The **name** argument is a string that identifies the text frame in question. The remaining arguments are internal data for special case use, such as when appending data to a newly created VP document.

The recommended usage is:

1) Enable **PO_COMPRESS** upon invoking **di_start()** or **di_startap()**. (This will cause paginate to fill the text in linked text frames.)
2) Append all of the text in the linked-text frame to the first link-order text frame. Internal data may be set to :

```
partab     = 1;
newpara    = FALSE;
newline    = FALSE;
paratab    = FALSE;
```

3) Append the text-link to the document via a call to **di_aptotxtlink()**.
4) Call **di_finish()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_aptotxtlnk() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumtxtlnk(), di_cleartxtlnk()

di__clearfillin

NAME

di__clearfillin - clear fill-in order

SYNOPSIS

```
#include "DocIC.h"

int
di_clearfillin(doc)
   di_doc doc;
```

DESCRIPTION

The **di_clearfillin()** function is used to cancel the previously specified fill-in order of an entire document. The **di_clearfillin**() function cancels the fill-in order previously specified. The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_clearfillin() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_aptofillin(), di_enumfillin()

di_cleartxtlnk

NAME

di_cleartxtlnk - clear text link

SYNOPSIS

```
#include "DocIC.h"

int
di_cleartxtlnk(doc)
    di_doc doc;
```

DESCRIPTION

The **di_cleartxtlnk()** function is used to clear the text frame link order of a document. This function is usually called in preparation of setting the text link order via **di_aptotxtlink()**.

The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_cleartxtlnk() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_aptotxtlink()

di__close

NAME

di__close - close a document

SYNOPSIS

```
#include "DocIC.h"

int
di_close(docptr)
   di_doc *docptr;
```

DESCRIPTION

The **di_close()** function is used to release the document handle of an enumerated document. Releasing the document handle frees the storage space originally allocated to it and sets the handle to **NULL**. The **doc** argument is the document handle to be terminated.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_close() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_open(), di_enumerate()

di_enumerate

NAME

di_enumerate - parse contents of a document

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_enumerate(to, procs, cdat, mrgnum,ret)
   di_tcont *to;
   di_enumprocs *procs;
   void *cdat;                  /* NULL */
   dp_bool mrgnum;              /* FALSE */
   dp_bool *ret;                /* Returned */
```

CALLBACK PROCEDURE

```
dp_bool
di_docproc(cdat, foprops, prprops, pgprops, lhead, rhead, lfoot, rfoot, num)
   void *cdat;
   dp_fontprops *foprops;
   dp_paraprops *prprops;
   dp_pageprops *pgprops;
   di_heading lhead;
   di_heading rhead;
   di_footing lfoot;
   di_footing rfoot;
   di_numbering num;

dp_bool
di_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
di_breakproc(cdat, foprops, brprops)
   void *cdat;
   dp_fontprops *foprops;
   dp_breakprops *brprops;
```

```
dp_bool
di_fieldproc(cdat, foprops, fiprops, field)
    void *cdat;
    dp_fontprops *foprops;
    dp_fldprops *fiprops;
    di_field field;

dp_bool
di_fntileproc(cdat, foprops)
    void *cdat;
    dp_fontprops *foprops;

dp_bool
di_indexproc(cdat, foprops, ixprops, index)
    void *cdat;
    dp_fontprops *foprops;
    dp_indexprops *ixprops;
    di_index index;

dp_bool
di_newparaproc(cdat, foprops, prprops)
    void *cdat;
    dp_fontprops *foprops;
    dp_paraprops *prprops;

dp_bool
di_pfcproc(cdat, foprops, pgprops, lhead, rhead, lfoot, rfoot, num)
    void *cdat;
    dp_fontprops *foprops;
    dp_pageprops *pgprops;
    di_heading lhead;
    di_heading rhead;
    di_footing lfoot;
    di_footing rfoot;
    di_numbering num;

dp_bool
di_sfbrkproc(cdat, num)
    void *cdat;
    dp_pagenumber num;

dp_bool
di_textproc(cdat, foprops, text)
    void *cdat;
    dp_fontprops *foprops;
    XString text;
```

DESCRIPTION

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

The **di_tcont** argument is to contain the file handle returned by an earlier call to **di_open()**. Refer to **di_apaframe()** for a description of **di_tcont**.

The **cdat** argument is a pointer to any user-defined data that is passed to the call-back procedure(s) specified in the **di_enumprocs** argument.

The **mrgnum** argument is short for "merge numbering". It is a Boolean value that, when set to **TRUE**, indicates that a page numbering pattern will be included in the heading or footing during enumeration. Setting this value to **TRUE** will result in the corresponding **di_numbering** in **di_pfcproc** and **di_docproc** to be set to **NULL.**

The **di_enumprocs** argument is a structure that contains user-defined call-back procedures for enumerating objects in the specified file. The members of **di_enumprocs** are:

```
di_docproc *doc;
di_aframeproc *aframe;
di_breakproc *break;
di_fieldproc *field;
di_fntileproc *fntile;
di_indexproc *index;
di_newparaproc *newpara;
di_pfcproc *pfc;
di_sfbrkproc *sfbrk;
di_textproc *text;
```

Each call-back procedure specified in **di_enumprocs** uses the properties and contents of the structure as parameters when invoked. The storage resources allocated to the properties passed to these functions is temporary; the user must explicitly copy any properties he or she may wish to save.

If **doc** is not **NULL, di_docproc()** will be called first with the first **foprops, prprops,** and **pgprops** present in the document. If **doc** is **NULL, di_newparaproc()** will be called and then **di_pfcproc** will be called with the first **foprops, prprops,** and **pgprops** present in the document.

When calling **di_pfcproc(),** if the headers are identical on the left and right pages, only **lhead** will contain the heading; **rhead** must remain **NULL**. The same rule applies to **lfoot** and **rfoot.**

Each call-back procedure returns a Boolean value. Enumeration stops when a return value is **TRUE**.

Some of the call-back procedures require a text container handle as a parameter. The text container handle may be specified recursively in calls to **di_enumerate()** in order to extract the contents of that same text container. For example, **di_fieldproc** may call **di_enumerate()** with **field** as the text container in order to extract the contents of the field. **di_enumerate()** requires a text container of type **di_tcont. di_cont** contains a union of two members: **type** and **h. type** is to be set to **TC_FIELD** and **h** is to be set to the field that was passed by a call to **di_fieldproc.**

Any handle returned by a call-back procedure is read only, and is valid only during the invocation of the call-back procedure. The handle returned is automatically released after execution of the call-back procedure. When a **NULL** handle is returned, it means the corresponding object does not contain text.

The initial paragraph and page format characters in a text container are also enumerated. Thus, when copying an existing document into a new document, avoid copying the initial paragraph and page format characters of the existing document as you copy the remainder of its contents.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_enumerate() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_open(), di_textforaframe(), di_close()

di_enumfillin

NAME

di_enumfillin - enumerate fill-in order

SYNOPSIS

```
#include "DocIC.h"

int
di_enumfillin(doc, proc, cdat)
    di_doc doc;
    di_fillinproc *proc;
    void *cdat;                  /* NULL */
```

CALLBACK PROCEDURE

```
dp_bool
di_fillinproc(cdat, name, type)
    void *cdat;
    XString name;
    di_fillintype type;
```

DESCRIPTION

The **di_enumfillin()** function is used to enumerate the fill-in order of fields and tables.

The **doc** argument is a document handle that was returned by an earlier call to **di_open()** or **di_startap**().

The **proc** argument is a pointer of the type **di_fillinproc()**. It specifies a call-back procedure to be invoked once for each object in the fill-in order. The arguments passed to **proc** specify user-defined data, the name of the enumerated object and its type. **di_fillinproc** may return **TRUE** to halt the enumeration.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_enumfillin() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_aptofillin(), di_clearfillin()

di_enumstyle

NAME

di_enumstyle - enumerate style

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_enumstyle(doc, fstyleproc, pstyleproc, cdat)
    di_doc doc;
    di_fstyleproc *fstyleproc;
    di_pstyleproc *pstyleproc;
    void *cdat;                    /* NULL */
```

CALLBACK PROCEDURE

```
dp_bool
di_fstyleproc(cdat, props)
    void *cdat;
    dp_fstyleprops *props;

dp_bool
di_pstyleproc(cdat, props)
    void *cdat;
    dp_pstyleprops *props;
```

DESCRIPTION

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.

The **doc** argument is a document handle that was returned by an earlier call to **di_open()** or **di_startap()**.

The **fstyleproc** and **pstyleproc** arguments are pointers to **di_fstyleproc** and **di_pstyleproc**, respectively. These call-back procedures are invoked once for each object in the style.They are invoked at the onset of **di_enumstyle()**'s execution, and, if either call-back procedure returns **TRUE**, the document enumeration process is aborted. If **FALSE** is returned, the process continues until completed.

The **cdat** argument is user-defined data that is passed to **fstyleproc** and **pstyleproc**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_enumstyle() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_apfstyle(), di_appstyle()

di_enumtxtlnk

NAME

di_enumtxtlnk - enumerate text link

SYNOPSIS

```
#include "DocIC.h"

int
di_enumtxtlnk(doc, proc, cdat)
   di_doc doc;
   di_txtlnkproc *proc;
   void *cdat;                  /* NULL */
```

CALLBACK PROCEDURE

```
dp_bool
di_txtlnkproc(item, cdat)
   di_textlink *item;
   void *cdat;
```

DESCRIPTION

The **di_enumtxtlnk()** function is used to enumerate the link order of a text frame.

The **doc** argument is a document handle that was returned by an earlier call to **di_open()** or **di_startap()**. It contains the text link order and text frame in question. If the text-link order is not included, **di_txtlnkproc** will not be called.

The **proc** argument is a pointer of the type **di_txtlnkproc**. It contains a call-back procedure that is invoked at the onset of **di_enumtxtlnk()**'s execution, and, if it returns **TRUE**, the enumeration process is aborted. If **FALSE** is returned, the process continues until completed.

The **cdat** argument is user-defined data that is passed to **proc**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_enumtxtlnk() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_aptotxtlnk(), di_cleartxtlnk()

di_finish

NAME

di_finish - finalize the document

SYNOPSIS

```
#include "DocIC.h"

int
di_finish(doc, proc, cdat, ret)
    di_doc *doc;
    di_ckabortproc *proc;      /* NULL */
    void *cdat;                /* NULL */
    ret_fc *ret;               /* Returned */
```

CALLBACK PROCEDURE

```
dp_bool
di_ckabortproc(cdat)
    void *cdat;
```

DESCRIPTION

The **di_finish()** function is used to finalize the document and to release the document handle, **doc**.

The **doc** argument is the file handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

The **proc** argument is a pointer of the type **di_ckabortproc**. It is a user-defined call-back procedure which can be used to abort the document generation process. It is invoked at the onset of **di_finish()**'s execution, and, if **di_ckabortproc** returns **TRUE**, the document generation process is aborted. If **FALSE** is returned, the process continues until completed.

The **cdat** argument is user-defined data that is passed to **di_ckabortproc**.

di_finish() returns **ret_fc**, a structure comprised of the following members:

```
dsktp_docref ref;
di_fcstat stat;
```

The first member, **dsktop_docref**, is the reference handle of the newly created document. This handle may be passed as an argument to **dsktp_movedoc()** to place the document on the desktop or in a folder. The second member, **status**, indicates the success or failure of the operation. **status** may one of the following values:

FC_OK	No errors were encountered.
FC_ABORT	Was unable to complete the document.
FC_DSKSP, FC_VM, FC_UNKNOWN	The document is finished but left unpaginated.

The resulting document file is temporary. To make the file permanent, call the **dsktp_movedoc()** function. It will place the document on the desktop or in a folder. The document that **di_finish()** provides will be in paginated form if the appropriate pagination parameters were specified in the initial call **di_start()** or **di_startap()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_finish() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_start(), di_startap(), dsktp_movedoc()

di_getfieldfromname

NAME

di_getfieldfromname - extract the properties of a named field

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "XString.h"

int
di_getfieldfromname(doc, name, props)
    di_doc doc;
    XString name;
    dp_fldprops *props;
```

DESCRIPTION

The **di_getfieldfromname()** function is used to search for a named field and list the properties of that field,

The **di_doc** argument contains a document handle that was returned by an earlier call to **di_open()**, **di_start()** or **di_startap()**.

The **name** argument is a string that specifies the name of the field from which to extract properties.

The **props** argument is a pointer of the type **dp_fldprops**. It specifies a list of the field properties to be extracted from the named field.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_getfieldfromname() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

di__getfnprops

NAME

di__getfnprops - get footnote properties

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_getfnprops(doc, procs, cdat)
    di_doc doc;
    di_fnpropsproc *procs;
    void *cdat;                    /* NULL */
```

CALLBACK PROCEDURE

```
int
di_fnpropsproc(cdat, nmprops, frprops, tfprops, foprops, pattern)
    void *cdat;
    dp_fnnumprops *nmprops;
    dp_frameprops *frprops;
    dp_tframeprops *tfprops;
    dp_fontprops *foprops;
    di_text pattern;
```

DESCRIPTION

The **di_getfnprops()** function is used to obtain the footnote properties of the document.

The **doc** argument is a document handle that was returned by an earlier call to **di_open()** or **di_startap()**.

The **procs** argument is a pointer of the type **di_fnpropsproc**. It is a call-back procedure that is invoked with all the footnote properties in the specified document. **di_fnpropsproc** does not need to call **di_reltext()** to release the text handle.

The **cdat** argument is a pointer to user-defined data.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_getfnprops() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_setfnprops()

di_open

NAME

di_open - open a document

SYNOPSIS

```
#include "DocIC.h"
#include "Desktop.h"

int
di_open(ref, ret)
   dsktp_docref ref;
   ret_open *ret;            /* Returned */
```

DESCRIPTION

The **di_open**() function is used to obtain the handle of a specific file. The returned file handle may then be passed as an argument to **di_enumerate()**, a function used to extract the contents of a file.

The **ref** argument is the handle of the document to be opened and is of the type **dsktp_docref**. **ref** is the document reference handle returned by an earlier call to **dsktp_getdocref()**, **dsktp_copydoc()** or **dsktp_enumerate()**.

di_open() returns **ret_open**, a structure that contains the following members:

```
di_doc doc;
di_opstat status;
```

doc is a document handle that may be passed to **di_enumerate()**. **status** is a code whose value indicates the success of the operation. The returned status code may be one of the following:

OP_OK	No errors were encountered.
OP_MALFORM	The Document is inconsistent internally.
OP_INCOMP	The version of the Document Editor used to open a document is different than the version used to create it.
OP_NOTLOCAL	The document is not on the local workstation, so it cannot be opened.
OP_DSKSP	Available disk space is insufficient to open the document.
OP_VM	Available contiguous virtual memory is insufficient to open the document.
OP_BUSY	Another process is using the file (e.g. background pagination).
OP_PASSWD	The user has invalid or incorrect credentials for opening the document.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_open() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumerate(), di_close()

di_rel*

NAME

di_relcap, di_relfield, di_relfoot, di_relhead, di_relnum, di_relindex, di_reltext - release storage

SYNOPSIS

```
#include "DocIC.h"

int
di_relcap(cap)
  di_caption *cap;

int
di_relfield(field)
  di_field *field;

int
di_relfoot(foot)
  di_footing *foot;

int
di_relhead(head)
  di_heading *head;

int
di_relnum(num)
  di_numbering *num;

int
di_relindex(index)
  di_index *index;

int
di_reltext(text)
  di_text *text;
```

DESCRIPTION

These functions are used to terminate handles, thus releasing the resources assigned to the respective handle. The user must call **di_relcap()**, **di_relfield()**, **di_relfoot()**, **di_relhead()**, **di_relnum()**, **di_relindex()**, or **di_reltext()** to release the resources associated with a non-**NULL** handle obtained from any **di_ap*()** function.

After calling **di_rel*()**, the respective handle will be invalid. To help prevent the use of an invalid handle, each **di_rel*()** routine removes the pointer to the respective handle and then sets the handle itself to **NULL**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_rel*() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

di__setfnprops

NAME

di__setfnprops - set footnote properties

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_setfnprops(doc, nuprops, frprops, tfprops, foprops, ret)
    di_doc doc;
    dp_fnnumprops *nuprops;    /* NULL */
    dp_frameprops *frprops;    /* NULL */
    dp_tframeprops *tfprops;   /* NULL */
    dp_fontprops *foprops;     /* NULL */
    di_text *ret;              /* Returned */
```

DESCRIPTION

The **di_setfnprops()** function is used to set the footnote properties of a document.

The **doc** argument is a document handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

The **nuprops** argument is a pointer of the type **dp_fnnumprops**. It is a structure containing data used to control the numbering of footnotes across documents during pagination of a book or a shared book.

The **frprops** argument is a pointer of the type **dp_frameprops**. It is a structure containing data that specifies the values of footnote frame properties, such as border thickness, number of columns to span, and margin control.

The **tfprops** argument is a pointer of the type **dp_tframeprops**. It is a structure that specifies the text frame properties, such as orientation and name.

The **foprops** argument is a pointer of the type **dp_fontprops**. It is a structure that specifies the font properties to be used in the footnotes, such as font type, placement, and offset.

This function returns **di_text**, a handle that may be passed to other **di_ap*()** functions. The **di_text** handle must be released via **di_reltext()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_setfnprops() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_getfnprops(), di_reltext()

di_setmode, di_getmode

NAME

di_setmode, di_getmode, - set or get the mode of properties for the document

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_getmode(doc, props)
    di_doc doc;
    dp_modeprops *props;

int
di_setmode(doc, props, select)
    di_doc doc;
    dp_modeprops *props;
    dp_modesel select;
```

DESCRIPTION

These two functions are used, either, to get or to set the mode properties of a document. Mode properties are Boolean variables that, when set to **TRUE**, display the structure, non-printing characters, cover sheet, and prompt fields in a document. These functions may be called at any time during the document generation process.

The **di_doc** argument is the document handle that was returned by an earlier call to **di_start()** or **di_startap()**.

dp_modeprops is an argument that points to a structure containing four Boolean fields that indicates the different display characteristics of the document in question.

The **dp_modesel** argument is an array that is used to specify those display characteristics to be affected. When setting mode properties, only those properties designated by **TRUE** selections will be changed.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_setmode() and **di_getmode()** will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

di_setpara

NAME

di_setpara - set current paragraph properties

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_setpara(to, prprops)
   di_tcont *to;
   dp_paraprops *prprops;
```

DESCRIPTION

The **di_setpara()** function is used to modify the paragraph properties of paragraphs in a specific text container. This function may be called at any time. If it is called repeatedly in the same paragraph, only the most recent call will remain in effect.

The **di_tcont** argument is the handle to the text container whose paragraph properties are to be modified. The text container may be any **di_tcont** or document. Refer to **di_apaframe()** for a description of **di_tcont**.

The **di_paraprops** argument points to a structure containing the set of paragraph properties to be modified.

di_setpara() affects the entire current paragraph, including portions not yet appended at the time **di_setpara()** is called. The property changes are also applied to all subsequent paragraphs unless the user overrides the properties with new ones passed to **di_apnewpara()**, or by another call to **di_setpara()**.

Setting text container paragraph properties will result in an error if the text container in question does not contain at least one paragraph character. Although paragraph characters are added (as necessary) during calls to **di_ap*()**, calling **di_setpara()** before calling any **di_ap*()** function will result in an error. To avoid this situation, the user may simply call **di_apnewpara()** to ensure that the **di_tcont** does have a paragraph character. **di_ap*()** functions will add a new paragraph character only if there is none already present, thus avoiding any duplication.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_setpara() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_apnewpara()

di_start

NAME

di_start - begin creation of a new document

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_start(pagiops, whead, wfoot, wnum, ifoprops, iprprops, ipgprops, styledat, ret )
    di_pagiops pagiops;                /* PO_COMPRESS */
    dp_bool whead;                     /* FALSE */
    dp_bool wfoot;                     /* FALSE */
    dp_bool wnum;                      /* FALSE */
    dp_fontprops *ifoprops;            /* NULL */
    dp_paraprops *iprprops;            /* NULL */
    dp_pageprops *ipgprops;            /* NULL */
    di_styledata *styledat;            /* NULL */
    ret_sc *ret;                       /* Returned */
```

CALLBACK PROCEDURE

```
int
di_styleproc(style,cdat, fstyleproc, pstyleproc)
    di_style style;
    void *cdat;
    di_apfstyleproc *fstyleproc;
    di_appstyleproc *pstyleproc;

int
di_apfstyleproc(style,styleprops)
    di_style style;
    dp_fstyleprops *styleprops;

int
di_appstyleproc(style,styleprops)
    di_style style;
    dp_pstyleprops *styleprops;
```

DESCRIPTION

The **di_start()** function is called to initiate the document generation process. It is used to create an empty document with specific format attributes, such as pagination and margin size. It then returns a file handle that needs to be passed as an argument to related **di_ap*()** functions. **di_finish()** is called to terminate the document generation process initiated by **di_start()**.

The **pagiops** argument specifies the type of pagination the finished document is to have. It may have one of three possible values: **PO_COMPRESS, PO_SIMPLE,** and **PO_NONE.**

> **PO_COMPRESS** pagination provides all the outward signs of pagination, such as page format properties, and leaves the structure of the document in an optimized form. An optimized document occupies less disk and buffer space than an unoptimized document.

PO_SIMPLE pagination provides the outward signs of pagination but does not leave the document in an optimized form. Therefore, subsequent editing may be slower than it would be for documents paginated with **PO_COMPRESS**. The advantage of this option over **PO_COMPRESS** is that this option completes the pagination process slightly faster than does **PO_COMPRESS**.

PO_NONE skips the pagination process entirely, thus leaving the document in a raw form. Raw form means that the document is neither paginated nor optimized. This may result in slow editing and potential loss of data. This option is recommended for only very small documents. If the document is to be more than a few pages in length, the user must specify a **pagiops** value other than **PO_NONE** to avoid losing data.

The **whead**, **wfoot** and **wnum** arguments are Boolean values that,when set to **TRUE**, insert heading, footing, and numbering properties into the first page format character (PFC) of the document.

The **ifoprops, iprprops**, and **ipgprops** arguments specify the initial font, paragraph, and page properties of the document, respectively. If these arguments are left **NULL**, **di_start()** will use a default set of properties. Refer to **dp_*props** for more information regarding properties and their default values.

When specifying the field properties for the **ipgprops** argument, page margins must be set so that at least one inch is left for text. An inch is the equivalent of 72 points. For example, (left margin + right margin + 72 < = page width), and (top margin + bottom margin + 72 < = page height).

The **styledat** argument is a pointer of type **di_styledata**. It is a structure used to call the call-back procedure, **di_styleproc**. The call-back procedure specifies the font and paragraph style properties of the new document. The **styledat** argument applies only to the first new paragraph and page format characters in the document. **di_styledata** contains the following members:

```
di_styleproc *styleproc;
void *cdat;
```

If **styledat** is a non-**NULL** value, the user-defined call-back procedure will be called before a document handle is returned.

Another way to add font and paragraph style properties is by calls to **di_apfstyle()** and **di_appstyle()**, Their full names are AppendFontStyle and AppendParagraphStyle, respectively. Note that properties for the first new paragraph character and the page format character can be set only by the **styledat** argument, not by the **di_apfstyle()** or the **di_appstyle()** functions.

di_start() sets the return 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;
```

The **di_doc** handle returned represents the new document. The user should pass this handle to **di_ap*()** functions to add information to the document. The handle is later released by a call to **di_finish()**.

If the user releases the handle without calling a **di_ap*()** function, the resulting file will be a 1-page document containing a single new paragraph and page format character, with the initial font, paragraph, and page props as specified in **ifoprops**, **iprprops**, and **ipgprops**, respectively.

di_heading, **di_footing**, and **di_numbering** are heading, footing and numbering handles, respectively. They will be **NULL** unless the user specified **whead, wfoot** or **wnum** = **TRUE**. If the headings, footings or numbering are valid, the user should call various **di_ap*()** routines to add text and formatting information, and then later release each handle with a call to **di_relhead(), di_relfoot()** or **di_relnum()**.

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

SC_OK No errors were encountered.

SC_DSKSP There is not enough disk space to perform the operation.

SC_VM There is not enough contiguous virtual memory to create.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_start() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_finish(), di_ap*(), di_relhead(), di_relfoot(), di_relnum()

di_startap

NAME

di_startap - start appending

SYNOPSIS

```
#include "DocIC.h"
#include "Desktop.h"

int
di_startap(ref, pagiops, ret)
    dsktp_docref ref;
    di_pagiops pagiops;            /* PO_COMPRESS */
    ret_startap *ret;              /* Returned */
```

DESCRIPTION

The **di_startap()** function is called to acquire a file handle that may be used by other **di_ap*()** procedures to append data to the end of an existing document.

The **ref** argument specifies the file that is to be opened. The **pagiops** argument specifies the type of pagination the appended data is to have. See **di_start()** for information regarding the construction of the **pagiops** argument.

ret_startap is returned and it contains the following members:

```
di_doc doc;
di_scstat status;
```

doc is a file handle for the document that is to have data appended.

status indicates the success of the **di_startap()** call. It may have any of the following values:

SC_OK	No errors were encountered.
SC_DSKSP	There is not enough disk space to perform the operation.
SC_VM	There is not enough contiguous virtual memory to create.
SC_BUSY	Another process is accessing the file.

When appending is complete, **di_finish()** must be called to release the doc handle. If the **status** returned is not **SC_OK**, then the doc handle will be **NULL** and **di_finish()** should not be called.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_startap() will fail if one or more of the following is true:

Doc_BadParm One of the specified arguments is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_start(), di_finish()

di_starttext

NAME

di_starttext - begin appending text

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_starttext(doc, frame, props, ret)
    di_doc doc;
    di_ins frame;
    dp_tframeprops *props;
    di_text *ret;
```

DESCRIPTION

The **di_starttext()** function is used to initiate the process of appending text to the body of an anchored text frame. **di_starttext()** readies an anchored text frame to accept new text, then returns an object handle which may be passed to any other **di_ap*()** operation. Once the data has been appended to the frame, the user should call **di_reltext()** with the text handle returned by **di_starttext()**.

The **doc** argument is the document handle returned by an earlier call to either **di_start()** or **di_startap()**. The **frame** argument is the frame handle returned by an earlier call to **di_apaframe()**. The **props** argument describes the text frame properties. Refer to DocICProps for more information regarding text frame properties.

It is not mandatory to call **di_starttext()** after calling **di_apaframe()**. Failure to call **di_starttext()** will only result in an empty text frame. The frame will be entirely empty except for the presence of one new paragraph character that has default paragraph and font properties.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_starttext() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_apaframe(), di_reltext()

di_textforaframe

NAME

di_textforaframe - retrieve text from an anchored frame

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"

int
di_textforaframe( cont, props, ret)
    di_ins cont;
    dp_tframeprops *props;
    di_text *ret;                    /* Returned */
```

DESCRIPTION

The **di_textforaframe()** function is used to extract text from an anchored frame during enumeration. The contents of the text handle returned by this function may be enumerated by supplying the text handle as an argument to **di_enumerate()**. After enumeration, call **di_reltext()** to release the text handle.

The **cont** argument is an instance of an anchored frame. This instance is supplied as an argument to the **di_aframeproc** call-back procedure.

The **props** argument is a pointer of the type **dp_tframeprops**. It is a structure that specifies a set of text frame properties. Text frame properties, such as name and description, are used to identify the frame in question. Since the text container passed from **di_aframeproc** is not unique for each enumeration, the instance handle alone cannot be used to identify the frame in question.

The frame to be enumerated cannot be in a document to which any object has been appended.This means that the frame instance that is returned by a call to **di_aframeproc** cannot be used be passed as the container to **di_textforaframe()**. To append an object to the frame that is returned by **di_enumerate(di_aframeproc())**:

1) Enumerate the source text frame via a call to **di_textforaframe()**.
2) Initialize the frame to which the text is to be appended via a call to **di_starttext()**.
3) Enumerate the source text and append it to the target frame via a call to **di_textproc(**call-back**)**.
4) Release the text handles returned via calls to **di_reltext()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

di_textforaframe() will fail if one or more of the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumerate(), di_reltext()

2. Document IC Property Library

dp_intro

NAME

dp_intro - introductory explanation of Document IC properties

DESCRIPTION

This library contains functions and data types used to describe document-related properties. The properties described below contain information that applies to all ***IC** interfaces.

Break Properties

The chief type in this section is **dp_breakprops**. It describes the properties of the page break character. **dp_breakprops** contains the following member:

```
dp_breaktype type;
```

dp_breaktype may have one of the following values:

```
BR_NPAGE        /* new page */
BR_NLPAGE       /* new left page */
BR_NRPAGE       /* new right page */
BR_NCOL         /* new column */
```

Field Properties

The chief field property is **dp_fldprops**. It describes the properties of a field character. **dp_fldprops** contains the following members:

```
dp_lang lang;
unsigned length;
dp_bool req;
dp_skpchoice skpif;
dp_bool stpskp;
dp_fldchoice type;
XString fillin;
XString desc;
XString format;
XString name;
XString range;
XString skpiffld;
dp_fontruns *fillinruns;
```

lang is the value of **dp_lang**, an enumerated type used to specify the alphabet that will be used, based upon nationality, to generate text in the date and amount fields.

length specifies the maximum number of logical characters the field may contain.

req specifies whether the user is required to fill in the field being generated. If **req** is **TRUE**, the user will not be able to use **NEXT** or **SKIP** to advance to the next field until this field has been given a value.

skpif specifies the conditions under which the user may press either the **NEXT** or **SKIP** button to skip the field. **stpskp** specifies the conditions under which the **NEXT** or **SKIP** buttons will be disabled. **skpif** may have one of the following values:

```
SKP_EMPTY         /* skip if the field is empty */
SKP_NOTEMPTY      /* skip if the field is not empty */
SKP_NEVER         /* never */
SKP_ALWAYS        /* always */
```

type is the value of **dp_fldchoice**, an enumerated type that specifies the kind of data to be placed in the field. It may have one of the following values:

```
FLD_ANY           /* any */
FLD_TEXT          /* text */
FLD_AMOUNT        /* amount */
FLD_DATE          /* date */
```

FLD_ANY indicates that the field may contain any characters, including frames (but not other fields). **FLD_TEXT** indicates that the field may contain only letters, digits, and symbols entered from the keyboard. **FLD_AMOUNT** indicates that the field may contain only numbers, spaces, and the following symbols: +_ * $, . (). **FLD_DATE** specifies that entries in the field may contain only a date.

fillin defines the fill-in rule for this field.

If the document is set to prompt for data to go in fields upon pressing the **NEXT** key, **desc** specifies the message that is to be displayed as the prompt.

format controls the format in which information is presented. It is affected by the value of **type**. For a **type** of **FLD_TEXT**, this property defines a required pattern that must be matched. For a **type** of **FLD_AMOUNT** or **FLD_DATE**, this field controls the form in which the contents of the field will be presented, regardless of how the user enters it. For a **type** of **FLD_ANY**, the format property will not be used.

name is the text name to be assigned to the field. If no name is provided, the field will automatically be named Field*n*, as in Field1, Field2, and so on.

range defines a specific range of acceptable entries. For example, if *A ctnl C* is specified, where *ctnl* is the control character, the *D* field may not be set and is skipped. Refer to the *Document Editor: Basics User Guide* for more information on **range**.

skpiffld contains the name of the field that will appear in the Field Properties sheet, *Skip if field.*

fillinruns is an auxiliary data structure that the user may attach to the **XString** that describes the fill-in rule for the field. A font run describes the subsequence of characters within an **XString** that share the same font attributes.

Font Runs

fillinruns is a pointer to **dp_fontruns**, a structure that permits the user to associate font properties with text. **XString** provides no facilities for associating font properties with text, therefore **DocICProps** has been designed to permit the user to create various font information structures that point into **XString**

structures. It is also possible to enumerate the font runs in a given **XString** body of text by a call to **dp_enumfrun**, but doing so requires that you know where the font runs are located or declare them yourself.

The data structures described here are used to mark font runs. A font run is defined as consecutive text characters sharing the same font. The members of **dp_fontruns** describe an array of font runs and an integer value that specifies the length of the array. **dp_fontruns** contains the following members:

```
unsigned short length;
dp_run *runs;
```

dp_fontruns points to **dp_run**, which is a structure containing an array of runs. **dp_run** is called to specify the beginning of a font run. **dp_run** contains the following members:

```
dp_fontprops props;
unsigned index;
```

props is the field describing the font used in the font run. **index** is the offset, specified in bytes, of the desired text within an array. A run is specified as the byte offset from the beginning of the byte array, as defined by **index**, to the byte after the byte run. For example:

```
XString = "ABCDEFGH"       (2 * 8 = 16 bytes)
    fontprops of ABC is font1
    fontprops of DE is font2
    fontprops of FGH is font3

thus:
    length will be 3
    runs[0].props will be font1
    runs[0].index will be 6      -- 3 characters (from 1 to 6)
    runs[1].props will be font2
    runs[1].index will be 10     -- 2 characters (from 7 to 10)
    runs[2].props will be font3
    runs[2].index will be 16     -- 3 characters (from 11 to 16)
```

Footnote Numbering Properties

The chief type in this section is the **dp_fnnumprops**, which describes the properties that affect numbering within a footnote. **dp_fnnumprops** contains the following members:

```
dp_numctrl numctrl;
dp_bool resteachpage;
dp_bool deferframes;
dp_bool rulingline;
dp_bool split;
dp_rulelen rulelen;
dp_indexrep indexrep;
dp_lang letters;
dp_replesent digits;
unsigned int otherrule ;
XString continuation ;
```

XString continued ;

numctrl is the value of **dp_numctrl**, an enumerated type that controls footnote numbering across documents during pagination of a book or shared book. **dp_numctrl** may have one of the following values:

```
NC_REST             /* restart */
NC_CONT             /* continue */
```

resteachpage is a Boolean value that determines whether the numbering of footnotes is to be set back to 1 for each new page or if footnote numbering is to continue in numeric sequence for all the pages in the document.

deferframes specifies whether the the body of text accompanying each footnote is to be placed on the same page as the corresponding footnote, or deferred so that all the footnote text bodies are placed at the end of the document.

rulingline specifies whether a ruling line is to be created.

split specifies whether split footnotes are to be created.

rulelen specifies the length of the ruling line. This option is enabled when the value of of **rulingline** is set to **TRUE**.

indexrep specifies the type of reference symbol to be used. It contains the following members and may have the corresponding values:

```
IR_INTEGER          /* integer */
IR_UPLETTER         /* upper case letter */
IR_LOWLETTER        /* lower case letter */
IR_DAGGERS          /* daggers */
```

letters specifies the alphabet to be used, based upon nationality. It may have one of the following values:

```
LANG_USE            /* USEnglish */
LANG_UKE            /* UKEnglish */
LANG_FRN            /* French */
LANG_GMN            /* German */
LANG_SWD            /* Swedish */
LANG_ITA            /* Italian */
LANG_DUT            /* Dutch */
LANG_DAN            /* Danish */
LANG_NOR            /* Norwegian */
LANG_FIN            /* Finnish */
LANG_SPN            /* Spanish */
LANG_POR            /* Portuguese */
LANG_JPN            /* Japanese */
LANG_FRCAN          /* FrenchCanadian */
LANG_ENCAN          /* EnglishCanadian */
```

digits specifies the manner in which numbers are displayed, based upon the respective numbering system. It may have the following value:

```
RP_ASCII            /* ASCII */
```

Font Properties

dp_fontprops is the chief type with respect to fonts. **dp_fontprops** contains the following members:

```
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;
```

The section titled *dp_fontdesc* describes the **fontdesc** field; the section titled *dp_props* describes the other fields in a **dp_fontprops**.

dp_fontdesc

dp_fontdesc contains the following members:

```
dp_family family;
dp_dvariant dvariant;
dp_weight weight;
unsigned short size;
```

family specifies the font that is to be used. It may have one of the following values:

```
FMY_CENT            /* century (also, classic)*/
FMY_FRUT            /* frutiger (also, modern) */
FMY_TITAN           /* titan */
FMY_PICA            /* pica */
FMY_TROJAN          /* trojan */
FMY_VINTAGE         /* vintage */
FMY_ELITE           /* elite */
FMY_LETTER          /* letter gothic */
FMY_MASTER          /* master */
FMY_CUBIC           /* cubic */
FMY_ROMAN           /* roman */
FMY_SCIENT          /* scientific */
FMY_GOTHIC          /* gothic */
FMY_BOLD            /* bold */
FMY_OCRB            /* ocrB */
FMY_SPOKES          /* spokesman */
FMY_XEROX           /* xerox logo */
FMY_CENTTHIN        /* century thin */
FMY_SCIENTTHIN      /* scientific thin */
FMY_HELV            /* helvetica */
FMY_HELVCOND        /* helvetica condensed */
FMY_OPTIMA          /* optima */
FMY_TIMES           /* times */
FMY_BASK            /* baskerville */
FMY_SPARTAN         /* spartan */
FMY_BODONI          /* bodoni */
```

FMY_PALATINO	/* palatino */
FMY_CALEDONIA	/* caledonia */
FMY_MEMPHIS	/* memphis */
FMY_EXCELSIOR	/* excelsior */
FMY_OLYMPIAN	/* olympian */
FMY_UNIVERS	/* univers */
FMY_UNIVERSCOND	/* univers condensed */
FMY_TREND	/* trend */
FMY_BOXPS	/* boxPS */
FMY_TERMINAL	/* terminal */
FMY_OCRA	/* ocrA */
FMY_LOGO1	/* logo1 */
FMY_LOGO2	/* logo2 */
FMY_LOGO3	/* logo3 */
FMY_GENEVA2	/* geneva2 */
FMY_TIMES2	/* times2 */
FMY_SQUARE3	/* square3 */
FMY_COURIER	/* courier */
FMY_FUTURA	/* futura */
FMY_PRESTIGE	/* prestige */
FMY_ALLGOTHIC	/* alLetterGothic */
FMY_SCHBOOK	/* century school book */

dvariant specifies the manner in which numeric characters are displayed, such as roman or italic. It may have one of the following values:

DV_ROMAN	/* roman */
DV_ITALIC	/* italic */

weight specifies the intensity at which characters are displayed. It may have one of the following values:

WT_MEDIUM	/* medium */
WT_BOLD	/* bold */

size is the size of the font. This value may be anywhere within the range of 0 to 1023, inclusive.

Other fields in dp_fontprops

udlines specifies the number of times that the character is to be underlined. Acceptable values range between 0 to 2, inclusive.

stkout specifies whether or not the character is to be struck horizontally through the middle.

place specifies the position of the character relative to the line. It may have one of the following values:

PL_NULL	/* null */
PL_SUB	/* subscript */
PL_SUBSUB	/* sub subscript */
PL_SUBSUP	/* sub superscript */
PL_SUP	/* superscript */
PL_SUPSUB	/* super subscript */
PL_SUPSUP	/* super superscript */

tobedel indicates that text has been marked for deletion in the Redlining mode.

revised indicates text that was typed while Redlining was enabled but was left unfinalized.

width specifies the spacing between characters in the Japanese character set. It may have one of the following values:

```
WD_PROP             /* proportional */
WD_QUARTER          /* quarter */
WD_THIRD            /* third */
WD_HALF             /* half */
WD_THREEQUART       /* three quarter */
WD_FULL             /* full */
```

Normal spacing is achieved by specifying **WD_PROP**.

stylename is a text string that specifies the name of the style sheet.

ntrelm specifies the neutral elements of a style property.

dp_fontelmarr controls subtle aspects of the text appearance. **dp_fontelmarr** is an array of **dp_bool** and may contain the following elements:

```
FE_FAMILY           /* family */
FE_DSGNVAR          /* design variant */
FE_WEIGHT           /* weight */
FE_PSIZE            /* point size */
FE_UDLINES          /* n underlines */
FE_STKOUT           /* strikeout */
F_PLACE             /* placement */
FE_TOBEDEL          /* to be deleted */
FE_REVISED          /* revised */
FE_WIDTH            /* width */
FE_TXTCOL           /* text color */
FE_HLCOL            /* highlight color */
```

An example of an array declaration is:

```
typedef dp_bool dp_fontelmarr[FE_HLCOL + 1];
```

The size of the preceding array is 12 ((FE_HLCOL = 11) +1), where **FE_FAMILY** is the first element and has a value of 0.

tranpare is a Boolean value that specifies whether the text will be displayed as a solid object or, if the text is placed over another object, the object in the background will show through the text.

txtcol and **hlcol** specify the color attributes of a text string. **txtcol** indicates the color of text which isn't highlighted. **hlcol** indicates the color of text which is highlighted. Any valid color may be specified.

Frame Properties

The chief type in this section is **dp_frameprops**. It specifies the properties to be attributed to an anchored frame. **dp_frameprops** contains the following members:

```
dp_borderstyle bdstyle;
unsigned bdthick;
dp_framedims frdims;
dp_bool fxw;
dp_boolfxh;
```

```
dp_span span;
dp_valignment valign;
dp_halignment halign;
unsigned short tmgn;
unsigned short bmgn;
unsigned short lmgn;
unsigned short rmgn;
dp_pagenumber pnum;
dp_bool tranpare;
dp_color bdcol;
dp_color bgcol;
```

bdstyle specifies the display characteristics of the lines comprising the frame border. It may have one of the following values:

```
BDS_INVISIBLE        /* invisible */
BDS_SOLID            /* solid */
BDS_DASHED           /* dashed */
BDS_BROKEN           /* broken */
BDS_DOTTED           /* dotted */
BDS_DOUBLE           /* double */
```

bdthick specifies the thickness of the frame border. This value is specified as an integer in units of points. A point is 1/72 inch.

bdthick is affected by the value of **bdstyle**. If **bdstyle** is set to **BDS_DOUBLE**, then **bdthick** may range from between 3 to 18, inclusive, in multiples of 3 points. The remaining values of **bdstyles** may have a **bdthick** value ranging from 1 to 6 points, inclusive.

frdims specifies the height and width of the frame. These dimensions are also in units of points, where one point is equivalent to 1/72 inch. **dp_framedims** contains the following members:

```
unsigned w;
unsigned h;
```

w is the width of the frame along the x axis. **y** is the height of the frame along the y axis.

fxw and **fxh** are Boolean values that, when set to **TRUE**, indicate whether the frame will expand when necessary and the direction of expansion. **fxw** permits expansion in a horizontal direction along the x axis. **fxh** permits expansion in a vertical direction along the y axis.

span specifies the amount of space the frame may occupy with respect to the page. **dp_span** may have one of the following values:

```
SP_FULCOLUMN         /* full column */
SP_FULPAGE           /* full page */
```

valign and **halign** are the values of **dp_valignment** and **dp_halignment**, respectively. They are used to control the alignment of the frame relative to the top and bottom edges of the page.

dp_valignment may have one of the following values:

```
VA_TOP               /* top */
VA_BOTTOM            /* bottom */
VA_FLOATING          /* floating */
```

dp_halignment can have any of the following values:

```
HA_LEFT            /* left */
HA_CENTERED        /* centered */
HA_RIGHT           /* right */
```

tmgn, bmgn, lmgn, and **rmgn** are the margins of the frame, expressed as points. One point is the equivalent of 1/72 inch.

pnum indicates the page number where the corresponding anchored frame resides. **dp_pagenumber** contains the following members:

```
unsigned relpn;
unsigned dispn;
```

relpn is the page number of the document, relative to the first page which resides at the start of the document. **dispn** is the property of the page format character which controls the display of page numbers.

Index Properties

The chief type in this section is **dp_indexprops**. It describes the properties of the Index option. **dp_indexprops** contains the following members:

```
dp_indexhdl sphdl;
dp_bool useclass;
dp_bool usealter;
XString class;
XString alter;
```

sphdl is the value of **dp_indexhdl**, an enumerated type that specifies the special handling that the index is to receive. This has the same effect as the Special Handling field in the Index Object Property Sheet. **sphdl** may have one of the following values:

```
IDX_UNIT           /* index as a unit */
IDX_IGNORE         /* ignore */
IDX_CLASSIFY       /* classify alike */
```

useclass is a Boolean value that indicates whether or not a classification is to be used. This has the same effect as the Use Classification field in the Index Object Property Sheet. A value of **TRUE** indicates that a classification is desired.

usealter is a Boolean value that specifies whether or not an alternate is to be used. This has the same effect as the Use Alternate Term field in the Index Object Property Sheet.

Page Properties

The chief type in this section is **dp_pageprops**, a structure that describes the various properties to be associated with a VP document page. **dp_pageprops** contains the following members:

```
/* layout properties */
    dp_pagedims dims;
    unsigned short tmgn;
    unsigned short bmgn;
    unsigned short lmgn;
    unsigned short rmgn;
    dp_pagesidestpagside;
    unsigned bindwidth;
/* column structure properties */
    unsigned ncol;
    dp_bool blcol;
    dp_bool uneqcol;
    unsigned short colsp;
    dp_colwidths *colwidths;
    dp_coldirct coldirct;
/* heading & footing properties */
    dp_hdfttype hdfttype;
    dp_bool hdthispage;
    dp_bool hdsamepage;
    dp_bool ftthispage;
    dp_bool ftsamepage;
    dp_horpos hdpos;
    dp_horpos ftpos;
/* page numbering properties */
    dp_pntype pagnumtype;
    dp_verpos vnum;
    dp_horpos hnum;
    unsigned stpagnum;
```

dims is the value of **dp_pagedims**, a structure that specifies the width and height of a document page in units of 1/72 inch. **dp_pagedims** contains the following members:

```
unsigned short w;
unsigned short h;
```

tmgn, **bmgn**, **lmgn**, and **rmgn** are integers that specify the page margins in units of 1/72 inch.

stpagside is the value of **dp_pageside**, an enumerated type that specifies whether or not the first, or starting, page of the document should be on the left-hand side or the right-hand side. **dp_pageside** may have one of the following values:

```
PS_NIL          /* nil */
PS_LEFT         /* left */
PS_RIGHT        /* right *
```

PS_NIL indicates that there is no difference between the left- and right-hand sides of a document.

bindwidth is the additional amount of space to remain on the left edge of the completed document to account for the space necessary during book binding.

ncol, **blcol**, **uneqcol**, and **colsp** determine column structure. **ncol** is an integer that specifies the number of columns per page. A maximum of 50 columns may be specified. **blcol** is a Boolean value that specifies whether the length of the column will be equal to the length of the page. **uneqcol** is a Boolean value that specifies whether the columns may have varying widths. **colsp** is the amount of space between columns, specified in units of 1/72 inch.

colwidths is a pointer to **dp_colwidths**, a structure that specifies the width of each column in a document. It contains the following members:

```
unsigned length;
dp_colwidth *widths;
```

length is an integer that specifies the number of columns. **widths** is a pointer to **dp_colwidth**, an integer that specifies the width of each column. The value of **widths** is specified in units of 1/72 inch. **dp_colwidth** contains the following member:

```
unsigned short w;
```

coldirct is the value of **dp_coldirct**, an enumerated type that specifies the direction of each column. It may have one of the following values:

```
CD_LR            /* left to right */
CD_RL            /* right to left */
```

hdfttype is the value of **dp_hdfttype**, an enumerated type that specifies how headings and footings in the PFC are to be propagated across pages. It may have one of the following values:

```
HFT_NONE         /* none */
HFT_CONT         /* continue */
HFT_RESET        /* reset */
```

The preceding are the same as those shown for Page Numbering in the Page Format Property Sheet and they accept the same values.

hdthispage is a Boolean value that determines whether the header is to be displayed on the current page or on the succeeding page. Page headers are enabled when a numbering pattern has been toggled in the PFC so that it is active and it's set to appear in the top margin. When the numbering pattern is active but set to appear on the bottom margin, **hdthispage** will have no effect.

ftthispage acts like **hdthispage** with respect to footers. See the previous paragraph.

hdsamepage is a Boolean value that determines whether the headers used on both the left and right pages will be identical.

ftsamepage acts like **hdsamepage** with respect to footers. See the previous paragraph.

hdpos and **ftpos** control the horizontal positioning of headers and footers, respectively. They may have one of the following values:

```
HP_LEFT          /* left */
HP_RIGHT         /* right */
HP_CENTERED      /* centered */
HP_OUTER         /* outer of page */
```

pagnumtype is the value of **dp_pntype**, an enumerated type that specifies the the type of PageNumbering to be used. It may have one of the following values:

```
PNT_NONE             /* none */
PNT_CONTNUM          /* continue only page number */
PNT_CONTNUMANDPAT    /* continue number and pattern */
PNT_RESTART          /* restart */
```

vnum and **hnum** are the values of **dp_verpos** and **dp_horpos**, respectively. They control the vertical and horizontal positioning of PageNumbering in the document. **vnum** may have one of the following values:

```
VP_TOP           /* top edge */
VP_BOTTOM        /* bottom edge */
```

hnum may have one of the following values:

```
HP_LEFT          /* left edge*/
HP_RIGHT         /* right edge*/
HP_CENTERED      /* center of page*/
HP_OUTER         /* left edge on left pages and right edge on  right pages*/
```

stpagnum is an integer value that specifies the page number to be assigned the starting page. All succeeding pages will incremented accordingly.

lnformat, and **lnloc** are currently not implemented.

Paragraph Properties

The chief type in this section is **dp_paraprops**. It is a structure that specifies the properties of paragraphs in the document It contains the following members:

```
dp_basprops basprops;
dp_tabstops tabstops;
XString stylename;
dp_paraelmarr ntrelm;
```

basprops is the value of **dp_basprops**, a structure that specifies the standard properties associated with every paragraph, such as justification, indentation, and language. These are the same properties that appear on the Paragraph property sheet. Refer to the section titled **dp_*intro** for more information on **dp_basprops**.

tabstops is the value of **dp_tabstops**, a structure that specifies the tab stops associated with paragraphs. These are the same properties that appear on the Tab Settings property sheet. Refer to sections titled *Basic Property Records* and *Tabs* for more information on **dp_tabstops**.

stylename is a text string that specifies the style name of paragraph property.

ntrelm is the value of **dp_paraelmarr**, an array of **dp_bool** that describes basic, or default, paragraph style properties. It is declared as follows:

```
typedef dp_bool dp_paraelmarr[PE_TABSTOPS + 1];
```

Individual elements may be assigned the following values:

```
PE_PRELEAD          /* pre leading */
PE_POSLEAD          /* post leading */
PE_LINDENT          /* left indent */
PE_RINDENT          /* right indent */
PE_LNH              /* line height */
PE_PARALIGN         /* para alignment */
PE_JUST             /* justified */
PE_HYPH             /* hyphenated */
PE_KPNEXT           /* keep with next para */
PE_LANG             /* language */
PE_STRSUC           /* streak succession */
PE_DEFTABLEAD       /* default tab stop dot leader */
PE_DEFTABJUST       /* default tab stop justified */
PE_DEFTABOFFSET     /* default tab stop offset */
PE_DEFTABALIGN      /* default tab stop alignment */
PE_TABSTOPS         /*  tab stops*/
```

Basic Property Records

dp_basprops contains the following members:

```
unsigned short prelead;
unsigned short poslead;
unsigned short lindent;
unsigned short rindent;
unsigned short lnh;
dp_paralign paralign;
dp_bool just;
dp_bool hyph;
dp_bool kpnext;
dp_lang lang;
dp_strsuc strsuc;
dp_deftabsp deftabsp;
dp_tabalign deftabal;
```

prelead and **postlead** are integers that specify the amount of space that is to precede and follow the paragraph, respectively. These values are specified in units of points, where 1 point is the equivalent to 1/72 inch.

lindent and **rindent** are integers that specify the amount of space that is to comprise the margins on the left and right sides of the paragraph, respectively. These values are specified in units of points, where 1 point is the equivalent to 1/72 inch.

lnh is an integer that specifies the height of lines comprising a paragraph. These values are specified in units of points, where 1 point is the equivalent to 1/72 inch.

paralign is the value of **dp_paralign**, an enumerated type that specifies how the paragraph is to be aligned relative to the containing text column or text block. It may have one of the following values:

```
PA_LEFT          /* left */
PA_CENTER        /* center */
PA_RIGHT         /* right */
```

just is a Boolean value that specifies whether the lines of text in paragraphs will be stretched to make the left and right edges consistently even. That is, the line will be justified. A value of **FALSE** will result in a ragged right edge.

hyph is a Boolean value that specifies whether words on the right side of a line that are too long to fit entirely on the one line should be hyphenated to facilitate justification. If justification is not enabled, this property will be ignored.

kpnext is a Boolean value that specifies whether, during pagination, the current paragraph is to be kept on the same page as the following paragraph.

lang is the value of **dp_lang**, an enumerated type that specifies the type of text characters that will be used in the paragraphs. The specified language is used in formatting decimal tabs, hyphenation, spell checking, and so. It may have one of the following values:

```
LANG_USE         /* USEnglish */
LANG_UKE         /* UKEnglish */
LANG_FRN         /* French */
LANG_GMN         /* German */
LANG_SWD         /* Swedish */
```

LANG_ITA	/* Italian */
LANG_DUT	/* Dutch */
LANG_DAN	/* Danish */
LANG_NOR	/* Norwegian */
LANG_FIN	/* Finnish */
LANG_SPN	/* Spanish */
LANG_POR	/* Portuguese */
LANG_JPN	/* Japanese */
LANG_FRCAN	/* FrenchCanadian */
LANG_ENCAN	/* EnglishCanadian */

strsuc is the value of of the type **dp_strsuc**, an enumerated type that specifies whether text characters should be generated within paragraphs from left to right (e.g. English) or right to left (e.g. Hebrew). **dp_strsuc** may have one of the following values:

SS_LR	/* left to right */
SS_RL	/* right to left */

deftabsp is the value of of the type **dp_deftabsp**, an unsigned number that specifies the default number of spaces between tab stops. The value is specified in units of points, where there 1 point is equal to 1/72 of an inch.

deftabal is the value of of the type **dp_tabalign**, an enumerated type that specifies the manner in which tabs are aligned relative to the left paragraph margin, the center of the paragraph, the right paragraph margin, or points. A point is the equivalent of 1/72 of an inch. **dp_tabalign** may have one of the following values:

TSA_LEFT	/* left */
TSA_CENTER	/* center */
TSA_RIGHT	/* right */
TSA_DECIMAL	/* decimal */

Tabs

dp_tabstop is an array of structures whose members specify the tab settings of the current paragraph. It contains the following members:

```
dp_bool dotld;
dp_bool eqsp;
dp_taboffset offset;
dp_tabalign align;
```

dotld is a Boolean value that specifies whether the tab will have leader dots.

eqsp is a Boolean value that specifies whether tabs will be equally spaced.

offset is the value of of the type **dp_taboffset**, an unsigned number that specifies the location of each tab stop, relative to the margin.

align is the value of the type **dp_tabalign**, an enumerated type that specifies the manner in which tabs are aligned relative to the left paragraph margin, the center of the paragraph, the right paragraph margin, or points. A point is the equivalent of 1/72 of an inch. **dp_tabalign** may have one of the following values:

```
TSA_LEFT            /* left */
TSA_CENTER          /* center */
TSA_RIGHT           /* right */
TSA_DECIMAL         /* decimal */
```

An array of tabstops used to create or modify an object in a document must be sorted by increasing order of **offsets**. An **offset** that is equal to the previous one is ignored. During enumeration, tabstop arrays passed to the user will always be sorted in this manner. The maximum number of tabstops that may be set in a paragraph is 100.

Document Mode Properties

Mode properties affect the auxiliary menus of a VP document. The key mode property is **dp_modeprops**. It contains the following members:

```
dp_bool strct;
dp_bool nonprint;
dp_bool cover;
dp_bool prompt;
```

strct, **nonprint**, **cover**, and **prompt** are Boolean values that specify the manner in which the document will be displayed. If set to **TRUE**, the document will display structure and non-printing characters, the cover sheet, and prompt fields, respectively.

dp_modesel specifies the **dp_modeelm** of a document to be manipulated. **dp_modesel** is an array of **dp_bool** and is declared as follows:

```
typedef dp_bool dp_modesel[ME_PROMPT + 1];
```

dp_modeelm is an enumerated type that may have one of the following values:

```
ME_STRCT            /* structure showing */
ME_NONPRINT         /* non printing showing */
ME_COVE             /* cover sheet showing */
ME_PROMPT           /* prompt fields */
```

Font Style Properties

The chief type in this section is **dp_fstyleprops**, a structure that specifies font style properties. **dp_fstyleprops** contains the following members:

```
dp_fontprops props;
XString desc;
unsigned short softpos;
unsigned short stylepos;
```

props and **desc** are the properties of the font style.

softpos is the position of the SoftKey used to invoke the stylesheet. **stylepos** is the position at which the stylesheet propertysheet is to appear on the Style Softkey Assignment Sheet. Please refer to the figure on the following page for more information on StyleSheet and Style SoftKey.

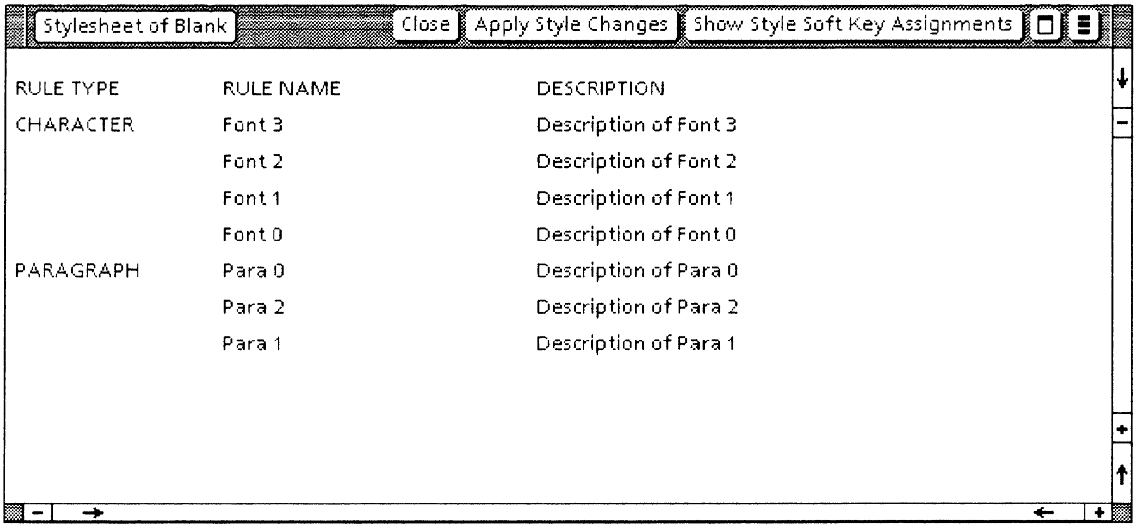
Paragraph Style Properties

The chief type in this section is **dp_pstyleprops**, a structure whose members specify the paragraph style properties. **dp_pstyleprops** contains the following members:

dp_paraprops props;
XString desc;
unsigned short softpos;
unsigned short stylepos;

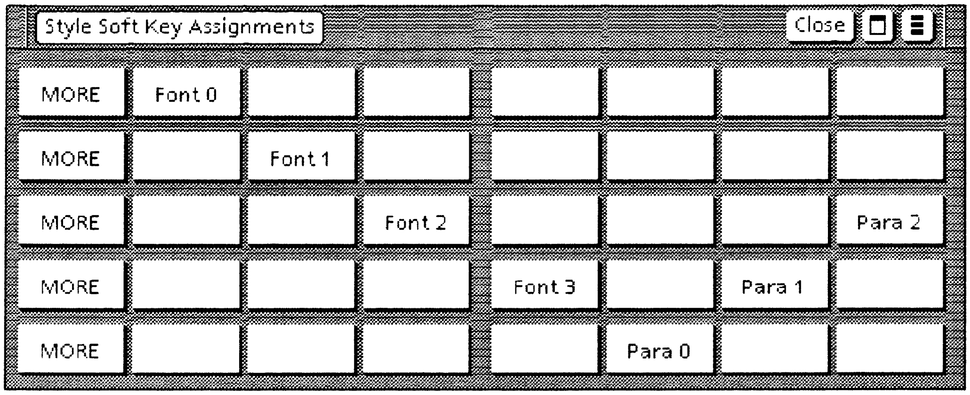
props and **desc** are the properties of the paragraph style.

softpos is the position of the SoftKey used to invoke the StyleSheet. **stylepos** is the position of the propertysheet of stylesheet. Please refer to the figure on the following page for more information on StyleSheet and Style SoftKey.



RULE TYPE	RULE NAME	DESCRIPTION
CHARACTER	Font 3	Description of Font 3
	Font 2	Description of Font 2
	Font 1	Description of Font 1
	Font 0	Description of Font 0
PARAGRAPH	Para 0	Description of Para 0
	Para 2	Description of Para 2
	Para 1	Description of Para 1

StyleSheet position from up to down corresponds to 0, 1, 2...
For example. Both the **stylepos** of Font 3 and Para 0 are 0.
Both the **stylepos** of Font 1 and Para 1 are 2.



MORE	Font 0						
MORE		Font 1					
MORE			Font 2				Para 2
MORE				Font 3		Para 1	
MORE					Para 0		

Style SoftKey position from left to right of 1st row corresponds to 1, 2...6, 7.
2nd row corresponds to 8, 9...13, 14. 3rd row...
For example. **softpos** of Font 1 is 9. **softpo** of Font 2 is 17. **softpos** of Para 0 is 33.

Position of StyleSheet and Style SoftKey

TextFrame Properties

The chief type in this section is **dp_tframeprops**, a structure whose members specify the text frame properties. **dp_tframeprops** contains the following members:

```
XString name;
unsigned innermargin;
dp_orient orientation;
dp_bool lastlinejust;
dp_bool autohyphen;
```

name is a text string that specifies the name of the text frame. **innermargin** is an unsigned number that specifies the amount of space to be allocated for the inner margin of all four edges of the frame. **innermargin** is specified in units of micas.

orientation is the value of dp_orient, an enumerated type that specifies the manner in which text is placed in the frame. Text may flow either horizontally (e.g., English) or vertically (e.g., Japanese). **dp_orient** can have any of the following values:

```
OR_HOR              /* horizontal */
OR_VER              /* vertical */
```

Only Japanese text may flow vertically.

lastlinejust is a Boolean value that, when set to **TRUE**, is used to specify whether the last line of text in linked text frames is to be justified.

autohyphen is a Boolean value that, when set to **TRUE**, is used to specify whether the last line of text in linked text frames is to be automatically hyphenated.

Color Properties

The chief type in this section is the **dp_color**, which describes the color properties. **dp_color** contains the following members:

```
int y;              /* 0 < = y < = 10000 */
int e;              /* -10000 < = e < = 10000 */
int s;              /* -10000 < = s < = 10000 */
```

The color is specified the combination of **y**, **e** and **s**, for example, black is specified as {0, 0, 0} and white is specified as {10000, 0, 0}. Refer to the Xerox *Color Encoding Standard* for more details.

dp_colorname is the name of the well known color which may have one of the following values:

```
CL_WHITE            /* white */
CL_BLACK            /* black */
CL_PINK             /* pink */
CL_RED              /* red *
CL_LGREEN           /* light green */
CL_GREEN            /* green */
CL_LBLUE            /* light blue */
CL_BLUE             /* blue */
CL_YELLOW           /* yellow */
CL_GOLD             /* gold */
CL_LORANGE          /* light orange */
CL_ORANGE           /* orange */
CL_VIOLET           /* violet */
CL_PURPLE           /* purple */
CL_TAN              /* tan */
CL_BROWN            /* brown */
CL_LGRAY            /* light gray */
CL_MGRAY            /* medium gray */
CL_DGRAY            /* dark gray */
CL_PGYELLOW         /* pale green yellow */
CL_LBYELLOW         /* light brilliant yellow */
CL_MYELLOW          /* moderate yellow */
CL_SYELLOW          /* strong yellow */
CL_PYELLOW          /* pale yellow */
```

```
CL_BYELLOW          /* brilliant yellow */
CL_MOYELLOW         /* moderate orange yellow */
CL_SOYELLOW         /* strong orange yellow */
CL_LOYELLOW         /* light orange yellow */
CL_DOYELLOW         /* deep orange yellow */
CL_LGYELLOW         /* light greenish yellow */
CL_GYELLOW          /* grayish yellow */
CL_POYELLOW         /* pale orange yellow */
CL_SORANGE          /* strong orange */
CL_MORANGE          /* moderate orange */
CL_SRORANGE         /* strong reddish orange */
CL_MRORANGE         /* moderate reddish orange */
CL_DRORANGE         /* dark reddish orange */
CL_VSRED            /* very strong red */
CL_BRED             /* brilliant red */
CL_MRED             /* moderate red */
CL_DAPRED           /* dark purplish red */
CL_SRED             /* strong red */
CL_MPRED            /* moderate purplish red */
CL_SPRED            /* strong purplish red */
CL_DRED             /* dark red */
CL_DEPRED           /* deep purplish red */
CL_VPRED            /* vivid purplish red */
CL_LYELLOW          /* light yellow */
CL_MYPINK           /* moderate yellow pink */
CL_PPPINK           /* pale purplish pink */
CL_DAPPINK          /* dark purplish pink */
CL_LPPINK           /* light purplish pink */
CL_DEPPINK          /* deep purplish pink */
CL_MPPINK           /* moderate purplish pink */
CL_GPPINK           /* grayish purplish pink */
CL_PPINK            /* pale pink */
CL_LRPURPLE         /* light reddish purple */
CL_VRPURPLE         /* vivid reddish purple */
CL_MRPURPLE         /* moderate reddish purple */
CL_SRPURPLE         /* strong reddish purple */
CL_DVIOLET          /* deep violet */
CL_MVIOLET          /* moderate violet */
CL_SVIOLET          /* strong violet */
CL_DAPBLUE          /* dark purplish blue */
CL_VPPBLUE          /* very pale purplish blue */
CL_LPBLUE           /* light purplish blue */
CL_SBLUE            /* strong blue */
CL_DEBLUE           /* deep blue */
CL_DEPBLUE          /* deep purplish blue */
CL_VLBLUE           /* very light blue */
CL_BBLUE            /* brilliant blue */
CL_DSBLUE           /* deep strong blue */
CL_DABLUE           /* dark blue */
CL_VPBLUE           /* very pale blue */
CL_VBLUE            /* vivid blue */
CL_DVBLUE           /* deep vivid blue */
CL_MBLUE            /* moderate blue */
CL_VLGBLUE          /* very light greenish blue */
CL_BGBLUE           /* brilliant greenish blue */
CL_SGBLUE           /* strong greenish blue */
CL_VGBLUE           /* vivid greenish blue */
```

```
CL_DGBLUE          /* deep greenish blue */
CL_VLGREEN         /* very light green */
CL_MBGREEN         /* moderate bluish green */
CL_SBGREEN         /* strong bluish green */
CL_DEBGREEN        /* deep bluish green */
CL_DABGREEN        /* dark bluish green */
CL_VPYGREEN        /* very pale yellow green */
CL_MGREEN          /* moderate green */
CL_DGREEN          /* deep green */
CL_BYGREEN         /* brilliant yellow green */
CL_VYGREEN         /* vivid yellow green */
CL_SYGREEN         /* strong yellow green */
CL_DYGREEN         /* deep yellow green */
CL_VPGREEN         /* very pale green */
CL_PYGREEN         /* pale yellow green */
CL_MBROWN          /* moderate brown */
CL_MRBROWN         /* moderate reddish brown */
CL_YWHITE          /* yellowish white */
CL_YGRAY           /* yellowish gray */
CL_PWHITE          /* purplish white */
CL_BWHITE          /* bluish white */
CL_LBGRAY          /* light bluish gray */
CL_BGRAY           /* bluish gray */
CL_DBGRAY          /* dark blueish gray */
CL_BBLACK          /* bluish black */
CL_VLGRAY          /* very light gray */
CL_VDGRAY          /* very dark gray */
```

dp__*col*

NAME

dp__colfromname, dp__namefromcol, dp__wkcolfromcol - color property

SYNOPSIS

```
#include "DocICProps.h"

int
dp_colfromname(name, ret)
   dp_colorname name;
   ret_wkcolfromname *ret;          /* Returned */

int
dp_namefromcol(color, ret)
   dp_color *color;
   ret_namefromwkcol *ret;          /* Returned */

int
dp_wkcolfromcol(color, ret)
   dp_color *color;
   ret_wkcolfromcol *ret;           /* Returned */
```

DESCRIPTION

The **dp_colfromname()** function is used to retrieve the integer equivalent of a well known color. The **name** argument is an integer value that specifies the name of the color. This function returns **ret**, a structure whose one member, **dp_color**, is an array of three integers that specifies the desired color. **ret** may then be passed as an argument to those functions that require color information.

The **dp_namefromcol()** function is used to retrieve the name of a color by supplying the data that defines the well known color. The **color** argument is a pointer to a structure whose three members define the color in question. This function returns **ret**, a structure containing the name of the color.

The **dp_wkcolfromcol()** function is used to retrieve a well known color from color. The **color** argument is a pointer to **dp_color**, a structure whose three members define that color. This function returns **ret**, a structure whose one member, **dp_color**, contains the integer data defining the well known color .

dp_color contains the following members:

```
int y;      /* 0 < = y < = 10000 */
int e;      /* -10000 < = e < = 10000 */
int s;      /* -10000 < = s < = 10000 */
```

color is specified as a combination of **y**, **e** and **s**. The number to color relationship is defined by the BWS framework. It is recommended that the user does not set the **y**, **e**, and **s** values directly. For example, black is specified as {0, 0, 0} and white is specified as {10000, 0, 0}. Note that **dp_color** may also be aliased by using **dp_yes**.

Refer to **dp_intro** at the beginning of this section for more information regarding colors.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dp_*col() will fail if one or more of the following are true:

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

dp__enumfrun

NAME

dp__enumfrun - enumerate font run

SYNOPSIS

```
#include "DocICProps.h"
#include "XString.h"

int
dp__enumfrun(r, runs, proc, cdat,ret)
    XString r;
    dp__fontruns *runs;
    dp__frunproc *proc;
    void *cdat;                /* NULL */
    dp__bool *ret;             /* Returned */
```

CALLBACK PROCEDURE

```
dp__bool
dp__frunproc(r, props, cdat)
    XString r;
    dp__fontprops *props;
    void *cdat;
```

DESCRIPTION

A font run is a way in which to associate font properties with text. The **dp__enumfrun()** function is used to enumerate user-defined fill-in runs, as defined in **dp__fldprops**. This is achieved by creating font information structures that point into associated XString structures.

The **r** argument is the text string to be enumerated. It is the value of the **fillin** argument to **dp__fldprops**.

The **runs** argument is a pointer to **dp__fontruns**, a structure whose members contain font properties and an index. It is the value of the **fillinruns** argument to **dp__fldprops**.

The **proc** argument is a pointer to **dp__frunproc**, a user-defined callback procedure. Its usage is defined by the user.

The **cdat** argument is user-defined data that is supplied to, and used by, **dp__frunproc** . Its usage is also defined by the user.

If **dp__frunproc()** returns **TRUE**, the enumeration stops and **ret** returns **TRUE**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dp_enumfrun() will fail if one or more of the following are true:

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

dp__get*def

NAME

dp__get*def - get default values of properties

SYNOPSIS

```
#include "DocICProps.h"

int
dp__getbreakdef(props)
    dp__breakprops *props;              /* Returned */

int
dp__getfielddef(props)
    dp__fldprops *props;                /* Returned */

int
dp__getfnnumdef(props)
    dp__fnnumprops *props;              /* Returned */

int
dp__getfontdef(props)
    dp__fontprops *props;               /* Returned */

int
dp__getfontdescdef(desc)
    dp__fontdesc *desc;                 /* Returned */

int
dp__getrundef(run)
    dp__run *run;                       /* Returned */

int
dp__getframedef(props)
    dp__frameprops *props;              /* Returned */

int
dp__getindexdef(props)
    dp__indexprops *props;              /* Returned */

int
dp__getpagedef(props)
    dp__pageprops *props;               /* Returned */

int
dp__getcolwidthdef(width)
    dp__colwidth *width;                /* Returned */

int
dp__getparadef(props)
    dp__paraprops *props;               /* Returned */

int
dp__getbaspropsdef(props)
```

```
    dp_basprops *props;              /* Returned */

int
dp_gettabstopdef(stop)
    dp_tabstop *stop;                /* Returned */

int
dp_getmodedef(props)
    dp_modeprops *props;             /* Returned */

int
dp_getfontstyledef(props)
    dp_fstyleprops *props;           /* Returned */

int
dp_getparastyledef(props)
    dp_pstyleprops *props;           /* Returned */

int
dp_gettframedef(props)
    dp_tframeprops *props;           /* Returned */

int
dp_getfontelmarralltrue(ret)
    dp_fontelmarr ret;               /* Returned */

int
dp_getparaelmarralltrue(ret)
    dp_paraelmarr ret;               /* Returned */

int
dp_getpagedel (ret)
    ret_getpagedel *ret;             /* Returned */

int
dp_gettoc(ret)
    ret_gettoc *ret;                 /* Returned */
```

DESCRIPTION

The **dp_get*def()** functions are used to obtain declared constants so that property structures may be initialized with neutral property values. A part of the information is obtained from the system defined data.

Before calling one of these functions, the user must declare a structure of the appropriate type and pass its address to the **dp_get*def()** function.

dp_getbreakdef() gets the following default values for page break properties:

```
    dp_breaktype type;           /* BR_NPAGE (new page) */
```

dp_getfielddef() gets the following default values for field properties:

```
dp_lang lang;                  /* LANG_USE (USEnglish) */
unsigned length;               /* 0 */
dp_bool req;                   /* FALSE */
dp_skpchoice skpif;            /* SKP_NEVER */
dp_bool stpskp;                /* FALSE */
dp_fldchoice type;             /* FLD_ANY */
XString fill-in;               /* NULL */
XString desc;                  /* NULL */
XString format;                /* NULL */
XString name;                  /* NULL */
XString range;                 /* NULL */
XString skpiffld;              /* NULL */
dp_fontruns *fillinruns;       /* NULL */
```

dp_getfnnumdef() gets the following default values for footnote numbering properties:

```
dp_numctrl numctrl;            /* NC_REST (restart) */
dp_bool resteachpage;          /* FALSE */
dp_bool deferframes;           /* FALSE */
dp_bool rulingline;            /* FALSE */
dp_bool split;                 /* FALSE */
dp_rulelen rulelen;            /* RL_ONETHIRD */
dp_indexrep indexrep;          /* IR_INTEGER */
dp_lang letters;               /* LANG_USE (USEnglish) */
dp_replesent digits;           /* RP_ASCII */
unsigned int otherrule ;       /* 144 */
XString continuation ;         /* NULL */
XString continued ;            /* NULL */
```

dp_getfontdef() gets the following default values for font properties:

```
dp_fontdesc fontdesc;
unsigned udlines;              /* 0 */
dp_bool stkout;                /* FALSE */
dp_place place;                /* PL_NULL */
dp_bool tobedel;               /* FALSE */
dp_bool revised;               /* FALSE */
dp_width width;                /* WD_PROP (proportional) */
XString stylename;             /* NULL */
dp_fontelmarr ntrelm;          /* all TRUE */
dp_bool tranpare;              /* TRUE */
dp_color txtcol;               /* 0, 0, 0 */
dp_color hlcol;                /* 10000, 0, 0 */
```

dp_getfontdescdef() gets the following default values for font description:

```
dp_family family;              /* FMY_FRUT (modern) */
dp_dvariant dvariant;          /* DV_ROMAN */
dp_weight weight;              /* WT_MEDIUM */
unsigned short size;           /* 12 */
```

dp_getrundef() gets the following default values for font run:

```
dp_fontprops props;
unsigned index;                /* 0 */
```

dp_getframedef() gets the following default values anchored frame properties:

```
dp_borderstyle bdstyle;        /* BDS_SOLID */
unsigned bdthick;              /* 2 */
dp_framedims frdims;           /* 72, 72 */
dp_bool fxw;                   /* TRUE */
dp_bool fxh;                   /* TRUE */
dp_span span;                  /* SP_FULCOLUMN (full column) */
dp_valignment valign;          /* VA_FLOATING */
dp_halignment halign;          /* HA_CENTERED */
unsigned short tmgn;           /* 18 */
unsigned short bmgn;           /* 18 */
unsigned short lmgn;           /* 0 */
unsigned short rmgn;           /* 0 */
dp_pagenumber pnum;            /* 1, 1 */
dp_bool tranpare;              /* FALSE */
dp_color bdcol;                /* 0, 0, 0 */
dp_color bgcol;                /* 10000, 0, 0 */
```

dp_getindexdef() gets the following default values for index properties:

```
dp_indexhdl sphdl;             /* IDX_UNIT (index as a unit) */
dp_bool useclass;              /* FALSE */
dp_bool usealter;              /* FALSE */
XString class;                 /* NULL */
XString alter;                 /* NULL */
```

dp_getpagedef() gets the following default values for page properties:

```
dp_pagedims dims;              /* 842, 595 */
unsigned short tmgn;           /* 72 */
unsigned short bmgn;           /* 72 */
unsigned short lmgn;           /* 72 */
unsigned short rmgn;           /* 72 */
dp_pageside stpagside;         /* PS_LEFT */
unsigned bindwidth;            /* 0 */
unsigned ncol;                 /* 1 */
dp_bool blcol;                 /* FALSE */
dp_bool uneqcol;               /* FALSE */
unsigned short colsp;          /* 18 */
dp_colwidths *colwidths;       /* NULL */
dp_coldirct coldirct;          /* CD_LR (left to right) */
dp_hdfttype hdfttype;          /* HFT_CONT (continue) */
dp_bool hdthispage;            /* TRUE */
dp_bool hdsamepage;            /* TRUE */
dp_bool ftthispage;            /* TRUE */
dp_bool ftsamepage;            /* TRUE */
dp_horpos hdpos;               /* HP_CENTERED */
dp_horpos ftpos;               /* HP_CENTERED */
dp_pntype pagnumtype;          /* PNT_NONE */
dp_verpos vnum;                /* VP_TOP */
dp_horpos hnum;                /* HP_RIGHT*/
```

```
unsigned stpagnum;            /* 1 */
```

dp_getcolwidthdef() gets the following default value of column width property:

```
unsigned short w;             /* 0 */
```

dp_getparadef() gets the following default values for paragraph properties:

```
dp_basprops basprops;
dp_tabstops tabstops;         /* 0, NULL */
XString stylename;            /* NULL */
dp_paraelmarr ntrelm;
```

dp_getbaspropsdef() gets the following default values for basic properties:

```
unsigned short prelead;       /* 0 */
unsigned short poslead;       /* 0 */
unsigned short lindent;       /* 0 */
unsigned short rindent;       /* 0 */
unsigned short lnh;           /* 12 */
dp_paralign paralign;         /* PA_LEFT */
dp_bool just;                 /* FALSE */
dp_bool hyph;                 /* FALSE */
dp_bool kpnext;               /* FALSE */
dp_lang lang;                 /* LANG_USE (USEnglish) */
dp_strsuc strsuc;             /* SS_LR(left to right) */
dp_deftabsp deftabsp;         /* 18 */
dp_tabalign deftabal;         /* TSA_LEFT */
```

dp_gettabstopdef() gets the following default values for tab stop:

```
dp_bool dotld;                /* FALSE */
dp_bool eqsp;                 /* FALSE */
dp_taboffset offset;          /* 0 */
dp_tabalign align;            /* TSA_LEFT */
```

dp_getmodedef() gets the following default values for mode properties:

```
dp_bool strct;                /* FALSE */
dp_bool nonprint;             /* FALSE */
dp_bool cover;                /* FALSE */
dp_bool prompt;               /* FALSE */
```

dp_getfontstyledef() gets the following default values for font style properties:

```
dp_fontprops props;
XString desc;                 /* NULL */
unsigned short softpos;       /* 0 */
unsigned short stylepos;      /* 0 */
```

dp_getparastyledef() gets the following default values for paragraph style properties:

```
dp_paraprops props;
XString desc;                 /* NULL */
unsigned short softpos;       /* 0 */
unsigned short stylepos;      /* 0 */
```

dp_gettframedef() gets the following default values for text frame properties:

```
XString name;                   /* NULL */
XString description;            /* NULL */
unsigned innermargin;           /* 141 */
dp_orient orientation;          /* OR_HOR (horizontal) */
dp_bool lastlinejust;           /* FALSE */
dp_bool autohyphen;             /* FALSE */
```

dp_getfontelmarralltrue() initializes all font elements properties to **TRUE**.

dp_getparaelmarralltrue() initializes all paragraph elements properties to **TRUE**.

dp_getpagedel() gets the XCCS code of the page number delimiter.

dp_gettoc() gets the XCCS code of the table of contents characters.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dp_get*def() will fail if one or more of the following are true:

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal..

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

3. Graphics IC Library

gi_intro

NAME

gi_intro - introductory explanation of graphics functions

DESCRIPTION

The functions in this section provide utilities for the creation and enumeration of anchored graphics, nested graphics, and CUSP button frames. The majority of these functions use Document IC Definitions and Document IC Property Definitions. Therefore, in addition to a familiarity with the Document Editor, you should also be familiar with these two sections of this manual before proceeding to use Graphics IC functions.

Creating Graphics

Graphics creation is initiated by a call to **gi_startgr()**. This function creates a frame in a document and returns an object called a **handle**. The resulting frame is a container in which may be placed graphics data, thus it is called a graphics container. A graphics container is defined as an object that can contain graphic objects and may be one of three basic types: an anchored graphics frame, a nested graphics frame, or a CUSP button within a graphics frame. The type of container it becomes is dependent upon the **gi_start*()** function that is called next, such as **gi_startnbtn()** or **gi_startcluster()**. Once a specific type of graphics container has been created, various **gi_ad*()** functions may be called to add graphic objects, such as curves, rectangles, bitmap graphics, and text frames

The **handle** is an opaque type that identifies the graphics frame in which will be placed graphics data and is, therefore, passed as an argument to the **gi_ad*()** functions.

A nested frame is a frame that is placed within a larger frame. Nested frames may be one of several types, such as non-anchored graphics frames, CUSP buttons, or graphics clusters. **gi_startgframe()**, **gi_startnbtn()**, or **gi_startcluster()** are called to create the corresponding nested frame. Each procedure takes a graphics container as an argument, and returns another graphics handle. The resulting handle is then passed as an argument to other **gi_ad*()** functions.

When everything has been added to a graphics container, the final step is a call to the respective **gi_finish*()** routine. These routines are **gi_finishgr()**, **gi_finishnbtn()**, **gi_finishgframe()**, or **gi_finishcluster()**. **gi_finishgr()** returns a graphics instance which can then be passed to **di_apaframe()**.

The typical scenario for creating a document with a floating graphics frame nested within an anchored graphics frame is as follows:

1. Call **di_create()** to obtain a document handle (**doc**).
2. Call **gi_startgr(doc)** to get an anchored frame handle (**h**).
3. Call **gi_ad*(h)** to add graphics to the anchored frame.
4. Call **gi_startgframe(h)** to get a handle for a nested graphics frame (**gfh**).
5. Call **gi_ad(gfh)** to add graphics to the nested frame.
6. Call **gi_finishgframe(gfh)** to finish the nested frame.
7. Call **gi_finishgr(h)** to complete the anchored frame and obtain an object of type **di_ins**.
8. Call **di_apaframe(h)**.
9. Call **di_finish(&doc)**.

Reading Graphics

There are also GraphicsIC functions that read the contents and properties of a graphics frame. The **gi_enumerate()** function is called to retrieve the contents or properties of a frame. It requires a graphics container and a set of user-defined call-back procedures as arguments. There is one call-back procedure for each type of object. Object types are defined as bar chart, bitmap frame, CUSP button, cluster, curve, ellipse, form field, graphics frame, line, line chart, pie chart, pie slice, point, rectangle, text, and triangle.

gi_enumerate() reads the contents of the graphics container, calling the appropriate procedure for each object type encountered. If a call-back procedure is not supplied for a particular type of object and that type of object is encountered during enumeration, that object will be ignored. Since call-back procedures are user-defined, they may be used to stop enumeration based upon a user-specified set of conditions.

Similarly, **gi_enumbtnprog()** accepts a set of user-defined call-back procedures to enumerate the contents of a CUSP button.

Cross References

The following pages contain charts that should be used to facilitate the selection and application of **gi_*()** functions. The charts are organized by category, or type of frame. When applicable, each category shows the types of objects that may be placed within the corresponding frame. The columns to the right of the categories list the functions that may be called to create an object or enumerate it.

Page numbers for each function may be found in either the table of contents or index.

Category of Anchored Graphics and Anchored Button Frames

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	

Category of 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

Category of Graphic Objects and Related Functions

Category	Objects	Creating	Reading
		Function Name:	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	

gi_adbacht

NAME

gi_adbacht - add bar chart

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adbacht(h, box, props, data, wchild, ret)
   gi_handle h;
   gi_box *box;                /* NULL */
   gi_bachtprops *props;       /* NULL */
   gi_chtdat *data;
   dp_bool wchild;             /* FALSE */
   gi_handle *ret;             /* Returned */
```

DESCRIPTION

The **gi_adbacht()** function is used to add a bar chart to a graphics container. This function draws a bar chart based on the properties specified by **gi_bachtprops**.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. It's two members, **place** and **dims** specify the origin of the bar chart and its size, relative to the graphics container.

```
gi_place place;
gi_dims dims;
```

gi_place contains two integer variables **x** and **y**. These two variables indicate the grid location of the box origin. **gi_dims** contains two integer variables **w** and **h**. These two variables indicate the width and height of the frame with respect to the box origin. Both **place** and **dims** are specified in units of micas.

A {0, 0} grid location indicates the upper-left corner of the frame. Increasing the value of **x** causes the placement location to shift towards the right. Increasing the value of **y** causes the placement location to shift downwards. It is illegal to specify negative **w** and **h** values

box.dims defines the size of the bar chart. Increasing the value of **w** causes the frame to grow towards the right. Increasing the value of **h** causes the frame to grow in a downward direction.

The **props** argument is a pointer of the type **gi_bachtprops**. It is a structure whose members specify the properties the resulting bar chart is to have. **gi_bachtprops** contains the following members:

```
double units;
unsigned div;
gi_barscale scale;
dp_color sclcol;
gi_balayout layout;
```

```
gi_baspacing spacing;
gi_baorient orient;
dp_bool key;
dp_bool bafloat;
dp_bool mirror;
gi_chtapps *apps;
dp_bool joined;
```

units, div, scale, sclcol, layout, spacing, orient, key, bafloat and **mirror** control some aspect of the bar chart's appearance. These members accept the same values as their counterparts in the bar chart property sheet.

units is a positive real number value that specifies the interval at which numeric indicators are placed on the scale. For example, a value of 2.5 means that all the numbers accompanying the scale will be divisible by 2.5. Therefore, only the numbers 2.5, 5.0, 7.5, etc. will be displayed.

div is a whole number between 0 and 65,535 that specifies the number of hash marks, or divisions, that are to occur between each numeric indicator on the scale.

scale is of the type **gi_barscale**. It is an enumerated variable that specifies the gauge to be used when displaying the bar chart. It may have one of the following values:

```
BS_STICK          /* single tick */
BS_DTICK          /* double tick */
BS_DGRID          /* double grid */
BS_OGRID          /* open grid */
```

sclcol is a structure of the type **dp_color**. It specifies the color to be used in drawing the bar chart scale.

layout is of the type **gi_balayout**. It is an enumerated variable that defines how the components comprising each bar in the chart is to be placed with respect to the other components. **layout** may have one of the following values:

```
BL_STACKED        /* place each component on top of the other component(s) */
BL_GROUPED        /* place components next to each other */
```

spacing is of the type **gi_baspacing** It is an enumerated variable that defines the separation between bar chart elements. It may have one of the following values:

```
BSP_MERGED        /* merged */
BSP_JOINED        /* joined */
BSP_QUARTER       /* quarter spacing */
BSP_HALF          /* half spacing */
BSP_THREEQUART    /* three-quarter spacing */
BSP_BRIDGED       /* bridged */
```

orient is of the type **gi_baorient**. It is an enumerated variable that defines the direction in which the bar chart data is to be drawn. The data may be drawn from the bottom of the frame to the top, or from the left edge of the frame to the right, **orient** may have one of the following values:

```
BO_VER            /* vertical */
BO_HOR            /* horizontal */
```

key is a Boolean value that, when set to **TRUE**, displays the explanatory notes in the bar chart.

bafloat is currently not supported.

mirror is currently not supported.

apps is a pointer of the type **gi_chtapps**. It is a structure that specifies the visual properties of the bars in the bar chart. It is used to define the color of the lines, the fill patterns, the color of the filled bars, etc. It contains the following members:

```
unsigned length;
gi_chtapp *values;
```

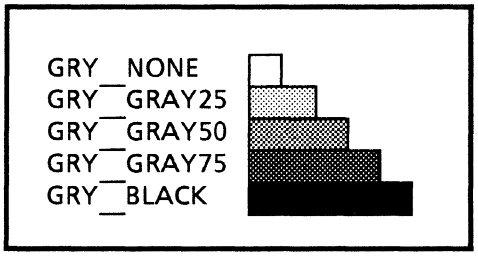
where, **gi_chtapp** contains the following members:

```
gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;
dp_color lncol;
```

gray is of the type **gi_gray** an enumerate type that specifies the amount of black, or saturation, to make varying shades of the color gray. It may have one of the following values:

GRY_NONE
GRY_GRAY25
GRY_GRAY50
GRY_GRAY75
GRY_BLACK

The number following the respective **GRY_GRAY*** indicates the percentage of saturation. For example,



txrs is of the type **gi_textures**. It is a structure that defines the direction in which the fill patterns are drawn in the resulting bars. It may have one of the following values:

dp_bool vertical
dp_bool horizontal
dp_bool nwse
dp_bool swne
dp_bool polkadot

txrcol, **shdcol**, and **lncol** are the respective colors of the fill pattern, the shading, and the lines used to draw each bar in the bar chart. **shdcol** is only available when **gray** is set to **GRY_BLACK**.

joined is a Boolean value that specifies whether the elements of the bar chart are to be merged as one with the bar chart, or if they are to remain separate graphic elements. If **joined** is **FALSE**, each graphics element, such as rectangles and lines, will be independent of the bar chart and may be manipulated accordingly.

data is a pointer of the type **gi_chtdat**. It is a structure whose members define the common data of the chart. It contains the following members:

```
XString title;
gi_dataset datset;
dp_lang lang;
gi_datsource datsou;
gi_labels *collabl;
gi_labels *rowlabl;
gi_datvalues *values;
```

title is of the type XString and is used to specify the name of the bar chart.

dataset is of the type **gi_dataset**. It is a structure that specifies the axis at which bar titles are to be drawn. It may have one of the following values:

```
DAS_COLUMN              /* column */
DAS_ROW                 /* row */
```

lang is of the type **dp_lang**, an enumerated variable that defines the language to be used in writing bar chart text. It may have one of fifteen values, such as **LANG_USE** or **LANG_JPN**. Refer to the section in *Document IC Property Definitions*, titled *Basic Property Records.* under the heading of **lang** for a description of acceptable values.

datsou is of the type **gi_datsource**, a structure that specifies the source that is to supply the data used to draw the individual bars of the bar chart. It contains the following members:

```
enum {
    DTS_PS,                 /* data in chart property */
    DTS_DOC                 /* data in document */
 } type;
union {
    gi_tblfillin fillin;    /* effective when type is DTS_PS */
    gi_tblcont doc;         /* effective when type is DTS_DOC */
    } u;
```

fillin is of the type **gi_tblfillin** and may have one of the following values:

```
TFO_BYROW               /* by row */
TFO_BYCOL               /* by column */
```

gi_tblcont contains the following members:

```
XString name;
gi_sousubset subset;
```

gi_sousubset contains the following members:

```
gi_elmrange colrange;
gi_elmrange rowrange;
```

gi_elmrange contains the following members:

```
unsigned first;
unsigned last;
```

The Document Editor may use two types of data from two different sources. One type and source of data is that from the chart property. The other is data from within a document. **DTS_PS** specifies that the source data for drawing bar charts is in the chart. **DTS_DOC** specifies that the source data for drawing the bars is in a table frame in the same document. If **DTS_DOC** is specified, **name** must also be specified. When data is supplied from a chart property, **gi_datsource** should be set as follows:

```
gi_datsource  datasource;

datasource.type = DTS_PS;
datasource.u.fillin = TFO_BYROW (or TFO_BYCOL);
```

When table data in a document is used as the source, **gi_datsource** should be set as follows:

```
gi_datsource  datasource;

datasource.type = DTS_DOC;
datasource.u.doc.name = (XString)tablename;
datasource.u.doc.subset.colrange.first = 0;
datasource.u.doc.subset.colrange.last = 0;
datasource.u.doc.subset.rowrange.first = 0;
datasource.u.doc.subset.rowrange.last = 0;
```

collabl and **rowlabl** are both pointers to **gi_labels**, a structure that specifies respective column and row bar titles. **gi_labels** contains the following members:

```
unsigned length;
XString (*values);              /* array of XString */
```

values is a pointer of the type **gi_datvalues**, a structure that specifies the values of text strings and numbers in the bar chart. It contains the following members:

```
enum {
   RS_STRING,
   RS_NUMERIC
    }format;
union {
   gi_strowcont string;         /* effective when format is RS_STRING */
   gi_numrowcont numeric;/* effective when format is RS_NUMERIC */
   } u;
```

gi_strowcont contains the following members:

```
   unsigned length;
   gi_strow *strow;             /* array of gi_strow */
```

gi_numrowcont contains the following members:

```
   unsigned length;
   gi_numrow *numrow;           /* array of gi_numrow */
```

strow is a pointer of the type **gi_strow**, a structure that contains an array of XString and its length.It represents the string data that is to be filled in the row. It contains the following members:

```
unsigned length;
XString *values;        /* array of XString */
```

numrow is a pointer of the type **gi_numrow**. It is a structure that contains an array of double and its length. It represents the numeric data to be filled in the row. It contains the following members:

```
unsigned length;
double *values;         /* array of double */
```

The data types **RS_STRING** or **RS_NUMERIC** are used as switches to select the elements of types, **string** or **numeric**.

wchild is a Boolean that, when set to **TRUE**, will cause a handle to the graphics elements in the bar chart to be returned in **ret**. After which, graphic elements may be added to the handle. When set to **FALSE**, **ret** will contain a **NULL** value and the Document Editor will rebuild the bar chart from the information contained in **gi_chtdat**. If a handle is returned, **gi_finishcht()** must be called to release it when done.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adbacht() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_finishcht()

gi_adbm

NAME

gi_adbm - add bitmap

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adbm(h, box, bmprops, frprops, wtcap, wbcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;                  /* NULL */
    gi_bmprops *bmprops;          /* NULL */
    gi_frameprops *frprops;       /* NULL */
    dp_bool wtcap;                /* FALSE */
    dp_bool wbcap;                /* FALSE */
    dp_bool wlcap;                /* FALSE */
    dp_bool wrcap;                /* FALSE */
    ret_adbm *ret;                /* Returned */
```

DESCRIPTION

The **gi_adbm()** function is used to add a bitmap graphic to the graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. Its two members, **place** and **dims**. specify the origin of the area in which the bit map will be placed and its size, relative to the graphics container (including caption area). Refer to **gi_adffield()** for a description of **gi_box**.

The **bmprops** argument is a pointer of the type **gi_bmprops**. It is a structure whose members control visual aspects of the bit map graphic. It contains the following members:

```
int xoffset;
int yoffset;
XString prntfile;
gi_bmdisp dispsou;
gi_bmscalprops scalprops;
dp_bool remotefile;
dp_color bitcol;
```

xoffset and **yoffset** have no affect on the outcome of a call to **gi_adbm()**.

prntfile is the full path name, or source, of the bitmap object to be printed. It is the means by which a different bitmap file may be accessed during the printing of the finished document than that being accessed when displaying the document. The value of this parameter is usually the same as the display source.

The source for the bitmap object to be placed in a document may be in one of two locations: either internal to the file (e.g., the bits are copied into the document), or in a file on the desktop (e.g., a pointer to the bits is inserted into the document). **dispsou** is of the type **gi_bmdisp**. It is a structure that specifies the display source of the bitmap object and whether the bitmap object is to be inserted or pointed to. **gi_bmdisp** contains the following members:

```
enum {
    BM_INTERNAL,
    BM_FILE
    } type;
union {
    gi_bmdat *bm;              /* effective when type is BM_INTERNAL */
    XString name;              /* effective when type is BM_FILE */
    } u;
```

The physical aspects of the actual bitmap object is described by the structure **gi_bmdat**. **gi_bmdat** contains the following members:

```
gi_rational xscl;
gi_rational yscl;
unsigned xdims;            /* # of bits wide */
unsigned ydims;            /* # of bits tall */
unsigned bpl;              /* Bits Per Line = ((xdim + 15) / 16) * 16 */
char *bitdata;
```

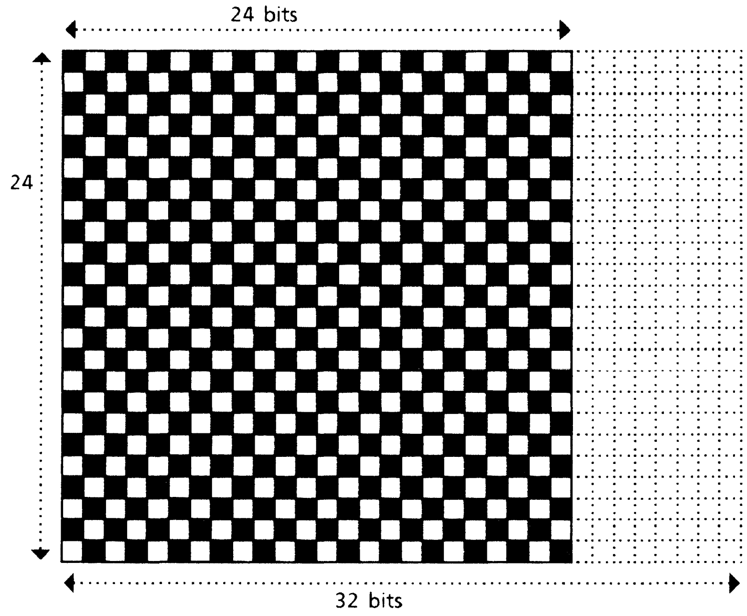
xscl and **yscl** are of the type **gi_rational**. It is a structure that specifies the scale at which to display the bitmap object in both x and y axis direction. **gi_rational** contains the following members:

```
int num;
unsigned den;
```

num and **den** are abbreviations of numerator and denominator, respectively. These two values are used to perform unit conversions from points to meters by specifying the resolution of the bitmap data. The base conversion involves converting dots per inch (dpi) into units of meters. For example, the desktop has a resolution of 72 dpi, therefore, for bitmap data created on the desktop, as one inch is equal to 0.0254 meters and there are 720,000 points in 254 meters, **xscl** and **yscl** should be set to {254, 720000}. If the bitmap data is created by a scanner, the resolution should be set to correspond to the resolution of the scanner. For example, if the scanner has a resolution of 200 dpi, then set **xscl** and **yscl** to {250, 200000}. If the resolution of the scanner is 300 dpi, then the correct values would be {254, 300000}.

xdims and **ydims** are unsigned integers that specify the x and y axis dimensions of the bitmap object in units of bits.

bpl is the real bitmap data per line. bpl must have a word boundary. For example:



```
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
10101010101010101010101000000000     5555H,  5500H,
01010101010101010101010100000000     AAAAH, AA00H,
10101010101010101010101000000000     5555H,  5500H,
01010101010101010101010100000000     AAAAH, AA00H,
10101010101010101010101000000000     5555H,  5500H,
01010101010101010101010100000000     AAAAH, AA00H,
10101010101010101010101000000000     5555H,  5500H,
01010101010101010101010100000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
10101010101010101010101000000000     AAAAH, AA00H,
01010101010101010101010100000000     5555H,  5500H,
```

```
xdims      =  24
ydims      =  24
bpl        =  32
bitdata    =  [
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H,
              AAAAH, AA00H, 5555H, 5500H, .....
```

bitdata is a pointer to to the bitmap data. The size of the bitmap data is to be equal to (**xdims** * **bpl**).

scalprops is a structure of the type **gi_bmscalprops**. It is used to specify the manner in which the bitmap is displayed. **gi_bmscalprops** contains the following members:

```
enum {
    BMS_PRNTRES,
    BMS_FIXED,
    BMS_AUTOMATIC
    } type;
union {
    unsigned res;              /* effective when type is BMS_PRNTRES*/
    gi_scalfix  fixed;         /* effective when type is BMS_FIXED */
    enum {                     /* effective when type is BMS_AUTOMATIC */
        SHP_SIMILAR,
        SHP_FILLUP
        } shape;
    } u;
```

gi_scalfix contains the following members:

```
enum {
    HAL_CENTER,
    HAL_RIGHT,
    HAL_LEFT
    } halign;
enum {
    VAL_CENTER,
    VAL_BOTTOM,
    VAL_TOP
    } valign;
unsigned   percent;
```

scalprops permits the user to specify one of three bitmap scaling modes: **BMS_PRNTRES, BMS_FIXED** or **BMS_AUTOMATIC.**

BMS_PRNTRES causes the bitmap object to be printed at the resolution specified in the **res** argument.

BMS_FIXED requires the user to control the bitmap's alignment (via **halign** and **valign** parameters) and scaling (via **xscl** and **yscl**). The printing of the bitmap object is also affected by the value of the **percent** argument. (See **percent** below.)

BMS_AUTOMATIC, with **shape** = **SHP_SIMILAR**, results in the bitmap object being enlarged or reduced to fit just inside the bitmap frame until either the vertical or horizontal edge of the bitmap object touches the graphic frame's edge. The aspect ratio of the bitmap object is maintained. This is usually the default mode. **BMS_AUTOMATIC**, with **shape** = **SHP_FILLUP**, results in the bitmap object being scaled to fit the entire graphic frame. The aspect ratio is not maintained.

If **BMS_PRNTRES** or **BMS_FIXED** is selected, **SHP_SIMILAR** and **SHP_FILLUP** will be ignored.

The **percent** parameter allows the user to shrink or magnify the bitmap object, while maintaining its aspect ratio. A **percent** value of 100 means that the bitmap should be displayed and printed the same size as the original. A value of 50 means that the bitmap is shrunk to one-half both vertically and horizontally. **percent** must be an integer ranging from 1 to 1000, inclusive. This parameter is only available when the value of **type** is set to **BMS_PRINTRES**.

The value of **res** specifies the resolution to be used in printing the bitmap object. It is usually set to the same resolution as the printer on which the bitmap object is to be printed. Standard values are 72, 75, 150, 200, and 300. Other values may be specified. Values are specified in units of dots-per-inch (dpi). This parameter is only available when the value of **type** is set to **BMS_PRINTRES**.

The **remotefile** parameter is used to specify whether the **prntfile** is in a file or on the desktop.

The **bitcol** parameter is a structure of the type **dp_color**. Its members describe the color of the dots. Refer to **dp_col*** for more information.

The **frprops** argument is a pointer of the type **gi_frameprops**. It is a structure that defines the common properties of the graphics frame. Refer to the description of **frprops** in **gi_startgframe()** for more information.

w*cap arguments are Boolean values that specify whether or not the frame is to have captions. If a value of **TRUE** is specified for a **w*cap** argument. the respective ***cap** return value will be non-**NULL**. These caption arguments are used to set the top, bottom, left, and right captions, respectively. Related **DocIC** functions may then be used to add text to each caption. Note that each caption must eventually be freed by a call to **di_relcap()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adbm() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

dp_namefromcol(), dp_wkcolfromcol(), gi_startgframe()

gi_adcurve

NAME

gi_adcurve - add curve

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adcurve(h, box, props)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_curveprops *props;           /* NULL */
```

DESCRIPTION

The **gi_adcurve()** function is used to add a curve of a specific size and shape to a graphics frame.

The **h** argument is the graphics frame handle returned by an earlier call to **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn**. **gi_startgr()**, or **gi_startcluster()**.

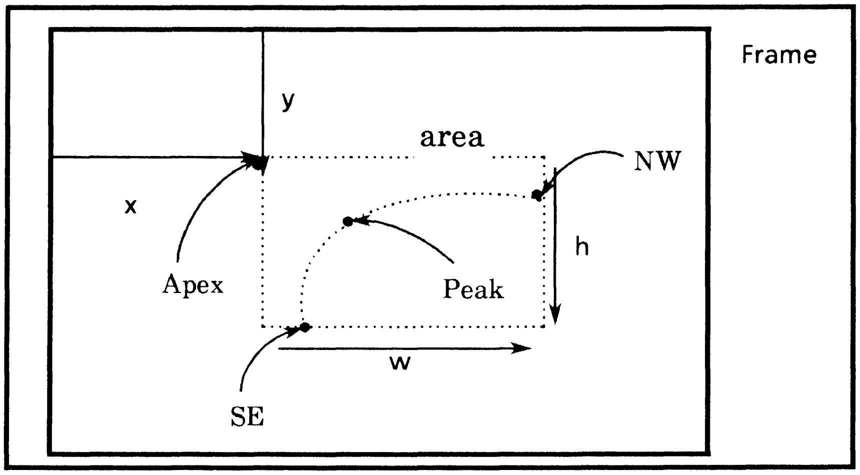
The **box** argument is a pointer of the type **gi_box**. It's two members, **place** and **dims** specify the origin of the object and its size, relative to the frame.

```
gi_place place;
gi_dims dims;
```

gi_place contains two integer variables **x** and **y**. These two variables indicate the grid location of the box origin. **gi_dims** contains two integer variables **w** and **h**. These two variables indicate the width and height of the box with respect to the box origin. Both **place** and **dims** are specified in units of micas.

A {0, 0} grid location indicates the upper left corner of a frame. Increasing the value of **x** causes the placement location to shift towards the right. Increasing the value of **y** causes the placement location to shift downwards. It is illegal to specify negative **w** and **h** values, therefore an object's **dims.place** must always correspond to the upper left corner of a box. It is legal to specify negative **x** and **y** values.

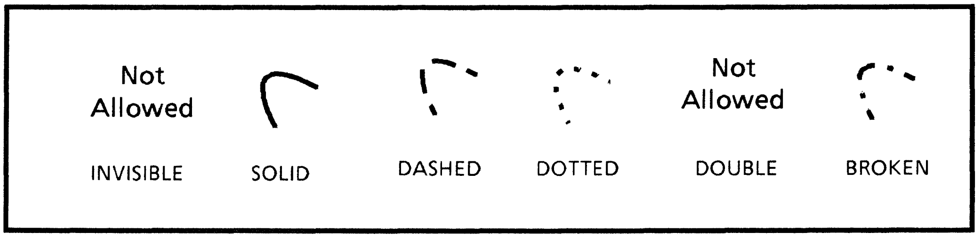
box.dims defines the area in which may be placed graphic objects. Increasing the value of **w** causes the frame to grow towards the right. Increasing the value of **h** causes the frame to grow in a downward direction.



The **props** argument is a pointer of the type **gi_curveprops**. It is a structure that defines the the appearance and shape of the curve. **gi_curveprops** contains the following members:

gi_brush brsh;
gi_lnend lnenw;
gi_lnend lnese;
gi_lnedhd lnhnw;
gi_lnedhd lnhse;
gi_place plnw;
gi_place plapx;
gi_place plse;
gi_place plpek;
dp_bool eccentric;
unsigned eccentricity;
dp_bool fixangle;

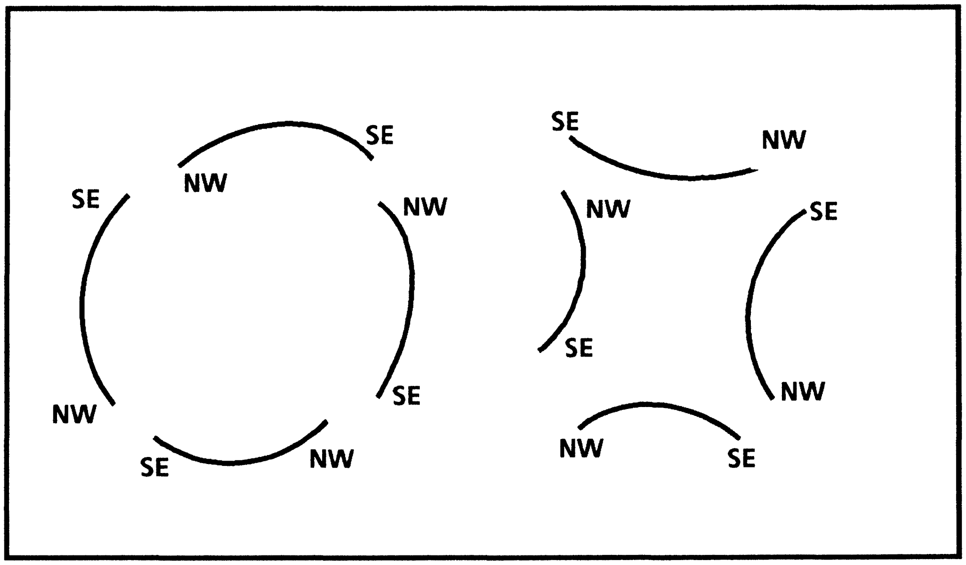
brsh is of the type **gi_brush**. It specifies the type of line used to draw the brush, such as solid or dashed, and the brush color. Refer to the description of **gi_startgframe()** for general information regarding **brsh**. The exception to the description of **brsh** in **gi_startgframe()** is with regards to the **stylebrush** member. The two parameters that may not be specified are **STB_INVISIBLE** and **STB_DOUBLE**. The remaining parameters will result in curves having the appearances as shown below:



lnenw and **lnese** are enumerated variables that describe the appearance of the end points of the curve. Each end point may have one of the following values:

LE_FLUSH	/* flush */
LE_SQUARE	/* square */
LE_ROUND	/* round */
LE_ARROW	/* arrowhead*/

lnenw defines the end that is painted first and **lnese** defines the end that is painted last. The curve is always traced in a clockwise direction, as shown in the figure below).



Defining Line Curves

If either **lnenw** or **lnese** is assigned a value of **LE_ARROW**, then the value of **lnhnw** and/or **lnhse** specifies the type of arrowhead to be placed at the endpoint(s) of the curve. Note that **lnhnw** specifies the type of arrowhead for **lnenw** and **lnhse** specifies the type of arrowhead for **lnese**.

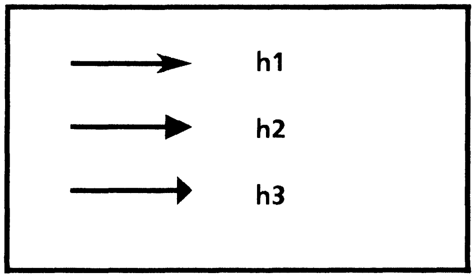
lnhnw and/or **lnhse** may have one of the following values:

LEH_NONE	/* none */
LEH_H1	/* h1 */
LEH_H2	/* h2 */
LEH_H3	/* h3 */

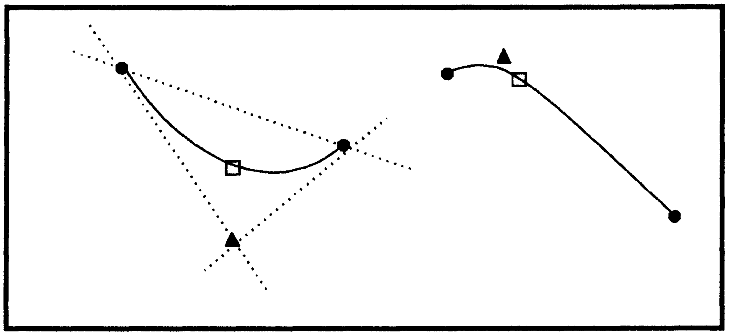
LEH_H1 is the finest point;**LEH_H3** is the most blunt, as shown in the figure below. If **lnenw** and/or **lnese** is not assigned a value of **LE_ARROW**, then **lnhnw** and/or **lnhse** should be left **LEH_NONE**.

The **pl*** parameters define the curve by specifying its end points, apex, and peak. These points are relative to the frame defined by the **box** argument, not the frame itself. Curves are traced in a clockwise direction, therefore, be sure that the NW endpoint appears before the SE endpoint when tracing a curve. The figure below illustrates the four **pl*** points used to define two different curves; the triangle marks the apex, the square marks the peak, and the circles mark the endpoints.

Another way to define a curve is by specifying the curve's endpoints, apex and **eccentricity**. **eccentricity** is a fraction used to specify the swell of a curve, as shown below.

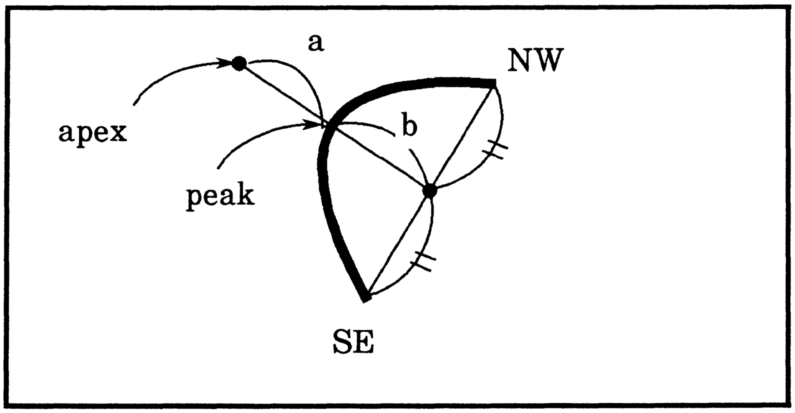


Types of Arrowheads



Defining Curves

The fraction is derived by the following equation:



Defining Eccentricity

eccentricity = b/(a + b) * 65535

The **eccentricity** argument is a Boolean value that, when set to **TRUE**, indicates that **eccentricity** is to be used rather than **pl*** points.

The **fixangle** parameter is a Boolean value that, when set to **TRUE**, indicates that the curve is to maintain its shape when grown or shrunk.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adcurve() will fail if one or more of the following are true:

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_startgframe()

gi_adellipse

NAME

gi_adellipse - add ellipse

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adellipse(h, box, props)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_ellipseprops *props;         /* NULL */
```

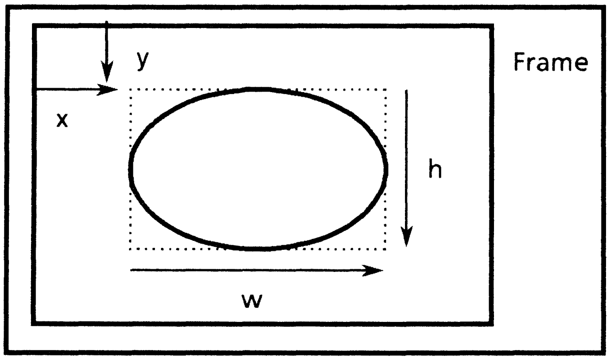
DESCRIPTION

The **gi_adellipse()** function is used to add an ellipse to a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_startgr()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. It's two members, **place** and **dims**. specify the origin of the box in which the ellipse will be placed and the area of the ellipse, relative to the graphics frame. Refer to **gi_adcurve()** for a description of **gi_box**.

The ellipse will be placed in the resulting box such that the extreme edges of the ellipse touch the respective edge of the box, therefore, the size of the box determines the size of the ellipse. For example,



The **props** argument is a pointer of the type **gi_ellipseprops**. It is a structure whose members define the appearance of the ellipse. Its members are:

```
gi_brush brsh;
gi_shading shade;
dp_bool fixshape;
```

brsh is a structure that defines the visual qualities of the lines used in tracing the border of the ellipse. It contains the following members:

```
unsigned wth;
gi_stlbrush stylebrush;
dp_color brushcolor;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

```
GSL W1          /* 1 width for Graphics Single Line */
GSL W2          /* 2 width for Graphics Single Line */
GSL W3          /* 3 width for Graphics Single Line */
GSL W4          /* 4 width for Graphics Single Line */
GSL W5          /* 5 width for Graphics Single Line */
GSL W6          /* 6 width for Graphics Single Line */
```

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

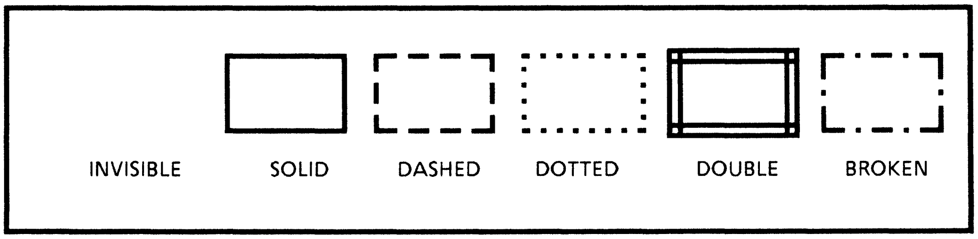
stylebrush defines how the lines are to be drawn, such as solid or dashed. It may have one of the following values:

```
STB_INVISIBLE   /* invisible */
STB_SOLID       /* solid */
STB_DASHED      /* dashed */
STB_DOTTED      /* dotted */
STB_DOUBLE      /* double */
STB_BROKEN      /* broken */
```

The **wth** of **STB_DOUBLE** borders is 3 times the usual width because it consists of two lines separated by a gap equal to the width of the line. In this case, the brush widths may have one of the following values:

```
GDL W1          /* 1 width for Graphics Double Line */
GDL W2          /* 2 width for Graphics Double Line */
GDL W3          /* 3 width for Graphics Double Line */
GDL W4          /* 4 width for Graphics Double Line */
GDL W5          /* 5 width for Graphics Double Line */
GDL W6          /* 6 width for Graphics Double Line */
```

Each value corresponds to 106, 212, 318, 423, 529, and 635 micas, respectively. The following are examples of brush styles:



brushcolor specifies the color to be used to display the lines that make up the edges of the graphic object. The value of color may be any color that is a member of **dp_color**.

shade is a structure of type **gi_shading**. It is used to define the appearance of the ellipse's interior. Its members are:

gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;

gray is of the type **gi_gray**, an enumerated variable that specifies the percentage of black, or saturation, to be used in making varying shades of the color gray. Refer to **gi_adbacht()** for a chart illustrating the available shades.

txrs is a structure of type **gi_textures**. It specifies the direction in which the texture is drawn in the ellipse or the type of texture that is to be placed in the ellipse. For example, textures may be placed in an ellipse with a horizontal, vertical, or diagonal orientation. Also, a type of texture that may be placed in the ellipse is a polka dot pattern. **gi_textures** has the following members:

dp_bool vertical;
dp_bool horizontal;
dp_bool nwse;
dp_bool swne;
dp_bool polkadot;

Each variable is a Boolean value. The resulting texture will be the AND of the variables . That is, each variable that is set to **TRUE** will be placed as a texture in the graphic object.

txrcol is a structure of type **dp_color**. Its members define the color that is to be used in drawing the texture, or foreground, of the ellipse's interior.

shdcol is a structure of type **dp_color**. Its members define the color to be used when drawing the background in the ellipse's interior. This parameter is enabled only when the value of **gray** is **GRY_BLACK**. If the value of **gray** is any other value, **shdcol** is set to **GRY_BLACK**.

fixshape is a Boolean value that, when set to **TRUE**, indicates that the aspect ratio of a graphic object will remain intact when the user grows or shrinks the ellipse. A value of **FALSE** indicates that the aspect ratio of the ellipse will change freely.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adellipse() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adbacht()

gi_adffield

NAME

gi_adffield - add form field

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adffield(h, box, fiprops, frprops, tfprops, paprops, foprops, wfield, wtcap, wbcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;                    /* NULL */
    dp_fldprops *fldprops;          /* NULL */
    gi_frameprops *frprops;         /* NULL */
    gi_tframeprops *tfprops;        /* NULL */
    dp_paraprops *paprops;          /* NULL */
    dp_fontprops *foprops;          /* NULL */
    dp_bool wfield;                 /* FALSE */
    dp_bool wtcap;                  /* FALSE */
    dp_bool wbcap;                  /* FALSE */
    dp_bool wlcap;                  /* FALSE */
    dp_bool wrcap;                  /* FALSE */
    ret_adffield *ret;              /* Returned */
```

DESCRIPTION

The **gi_adffield()** function is used to add a form field to a graphics frame.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_startgr()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. It's two members, **place** and **dims**. specify the origin of the frame and its size, relative to the graphics container.

```
gi_place place;
gi_dims dims;
```

gi_place contains two integer variables **x** and **y**. These two variables indicate the grid location of the box origin (including the caption). **gi_dims** contains two integer variables **w** and **h**. These two variables indicate the width and height of the frame with respect to the box origin. Both **place** and **dims** are specified in units of micas.

A {0, 0} grid location indicates the upper-left corner of the graphics container. Increasing the value of **x** causes the placement location to shift towards the right. Increasing the value of **y** causes the placement location to shift downwards. It is illegal to specify negative **w** and **h** values

box.dims defines the size of the frame. Increasing the value of **w** causes the frame to grow towards the right. Increasing the value of **h** causes the frame to grow in a downward direction.

Refer to **gi_startgframe()** for a description of the **box**, ***frprops**, and **w*cap** arguments.

The **fldprops** argument is a pointer of the type **dp_fldprops**. It is a structure whose members define the properties to be attributed to the resulting field. The members specify font properties, language, format, and so on. **dp_fldprops** has the following members:

```
dp_lang lang;
unsigned length;
dp_bool req;
dp_skpchoice skpif;
dp_bool stpskp;
dp_fldchoice type;
XString fill-in;
XString desc;
XString format;
XString name;
XString range;
XString skpiffld;
dp_fontruns *fillinruns;
```

Refer to **Field Properties** in the section *dp_intro* for a description of each parameter.

The **tfprops** argument is a pointer of the type **gi_tframeprops**. It is a structure whose members describe the properties of the text field and contains the following members:

```
dp_bool expr;
dp_bool expb;
dp_tframeprops props;
```

expr and **expb** are abbreviations for expand right and expand bottom, respectively. They are Boolean values. When both **expr** and **expb** are **TRUE**, the width and height can be changed according to the size of the text included.

The **props** argument is a pointer of the type **dp_tframeprops**. It is a structure whose members define the inner margin and orientation of the text within the frame, as well as the type of line justification and auto-hyphenation options. It contains the following members:

```
XString name;
XString description;
unsigned innerMargin;
dp_orient orientation;
dp_bool lastLineJustify;
dp_bool autoHyphenate;
```

Refer to **Text Properties** in **dp_intro** for a more thorough description.

The **paprops** and **foprops** arguments are pointers to **dp_paraprops** and **dp_fontprops**, respectively. They define the paragraph and font properties to be attributed to the resulting text field. See **Paragraph Properties** and **Font Properties** in the section *dp_intro* for a more complete description.

The **wfield** argument is a Boolean value that, when set to **TRUE**, causes **di_adfield** to return a handle to a field. The handle may then be passed as an argument to other text field manipulation functions. The **w*cap** arguments are Boolean values that specify if captions are desired along the top, bottom, left, or right edges of the text field.

This function sets the return information into the structure **ret_adffield**, which contains the following members:

```
di_field field;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
```

When **wfield** is set to TRUE, **gi_adffield()** will return **di_field**, a handle that may be used by other text field manipulation functions. This field handle must eventually be freed by a call to **di_relfield()**. Information may be added to this field by making calls to the respective **gi_ad*()** functions.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adffield() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_relfield()

gi_adline

NAME

gi_adline - add line

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adline(h, box, props)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_lineprops *props;            /* NULL */
```

DESCRIPTION

The **gi_adline()** function is used to add a line to a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_startgr()**, **gi_startbtn()**, **gi_startnbtn()**, **gi_startcluster()**, **gi_adpicht()**, **gi_adlncht()**, or **gi_adbacht()**.

The **box** argument is a pointer of the type **gi_box**. Refer to **gi_adcurve()** for a description of **gi_box**.

The **props** argument is a pointer of the type **gi_lineprops**. It is a structure whose members define the appearance and direction of the line. It contains the following members:

```
gi_brush brsh;
gi_lnend lnenw;
gi_lnend lnese;
gi_lnedhd lnhnw;
gi_lnedhd lnhse;
gi_lndirct dirct;
dp_bool fixangle;
```

Refer to **gi_adcurve()** for a description of the members of **gi_lineprops**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adline() will fail if one or more of the following are true:

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adcurve(), gi_adellipse()

gi__adlncht

NAME

gi__adlncht - add line chart

SYNOPSIS

```
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adlncht(h, box, props, data, wchild, ret)
    gi_handle h;
    gi_box *box;                /* NULL */
    gi_lnchtprops *props;       /* NULL */
    gi_chtdat *data;
    dp_bool wchild;             /* FALSE */
    gi_handle *ret;             /* Returned */
```

DESCRIPTION

The **gi_adlncht()** function is used to add a line chart to a specified graphics container.

Refer to **gi_adbacht()** for a description of the **h** and **box** arguments.

The **props** argument is a pointer of the type **gi_lnchtprops**. It is a structure whose members specify the properties of the resulting line chart. **gi_lnchtprops** contains the following members:

```
double xunits;
double yunits;
double xmax;
double xmin;
double ymax;
double ymin;
unsigned xdiv;
unsigned ydiv;
gi_axtype xaxtype;
gi_axtype yaxtype;
gi_rotation axorient;
dp_bool key;
dp_color scalcol;
gi_lnchtapps *apps;
dp_bool joined;
```

xunits, yunits, xmax, xmin, ymax, ymin, xdiv, ydiv, axorient, key and **scalcol** have the same range of values as their counterparts in the line chart property sheet.

xaxtype and **yaxtype** are of the type **gi_axtype**, an enumerated variable that specifies the gauge, or grid increments, to be used in generating the line chart. It may have one of the following values:

```
AXT_NONE        /* none */
AXT_SPLAIN      /* single plane */
AXT_STICK       /* single tick */
AXT_DPLAIN      /* double plane */
AXT_DTICK       /* double tick */
AXT_DFULL       /* double full */
```

axorient is of the type **gi_rotation**, an enumerated variable that specifies the orientation with which the chart and all its elements are to be inserted within the document. It may have one of the following values:

```
RT_NORMAL       /* normal */
RT_90           /* rotate 90 */
RT_180          /* rotate 180 */
RT_270          /* rotate 270 */
```

key is a Boolean value that, when set to **TRUE**, displays the explanatory notes in the line chart

sclcol is of the type **dp_color**. It is a structure that specifies the color to be used in drawing the line chart scale.

apps is of the type **gi_lnchtapps**. It is a structure that specifies the visual attributes of the lines used to draw the elements of the line chart itself, such as point size, fill pattern and brush. It contains the following members:

```
unsigned length;
gi_lnchtapp *values;
```

values is a pointer to an array of **gi_lnchtapp**. It is a structure that contains the following members:

```
unsigned psize;
gi_ptfill pfill;
gi_ptstyle pstyle;
dp_color pcolor;
gi_curvetype ctype;
gi_brush cbrush;
```

ctype is a structure of the type, **gi_curvetype**. It may have one of the following values:

```
CUT_STRAIGHT    /* straight */
CUT_SPLINE      /* spline */
CUT_BESTFIT     /* best fit straight */
CUT_EXP         /* exponential */
```

pfill and **pstyle** are of the type **gi_ptfill** and **gi_ptstyle**, respectively. They are described in **gi_adpoint()**.

joined is a Boolean value that specifies whether the elements of the line chart are to merged as one with the line chart, or if they are to remain separate graphic elements. If **joined** is **FALSE**, each graphics element, such as rectangles and lines, will be independent of the line chart and may be manipulated accordingly.

data is a pointer of the type **gi_chtdat**. See **gi_adbacht()** for a description of **gi_chtdat**.

wchild is a Boolean that, when set to **TRUE**, will cause a handle to the line chart to be returned in **ret**. After which, graphic elements may be added to the handle. When set to **FALSE**, **ret** will contain a **NULL** value and the document editor will build the line chart from the information contained in **gi_chtdat**. If a handle is returned, **gi_finishcht()** must be called to release it when done.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adlncht() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adpoint(), gi_adbacht(), gi_finishcht()

gi_adpicht

NAME

gi_adpicht - add pie chart

SYNOPSIS

```
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adpicht(h, box, props, data, wchild, ret)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_pichtprops *props;           /* NULL */
    gi_chtdat *data;
    dp_bool wchild;                 /* FALSE */
    gi_handle *ret;                 /* Returned */
```

DESCRIPTION

The **gi_adpicht()** function is used to add a pie chart to a specified graphics container.

See **gi_adbacht()** for a description of the **h** and **box** arguments.

The **props** argument is a pointer of the type **gi_pichtprops**. It is a structure whose members specify the properties of the resulting pie chart. **gi_pichtprops** contains the following members:

```
unsigned wth;
gi_piestyle style;
gi_chtapps *apps;
dp_bool joined;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

```
GSL W1          /* 1 width for Graphics Single Line */
GSL W2          /* 2 width for Graphics Single Line */
GSL W3          /* 3 width for Graphics Single Line */
GSL W4          /* 4 width for Graphics Single Line */
GSL W5          /* 5 width for Graphics Single Line */
GSL W6          /* 6 width for Graphics Single Line */
```

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

style is a structure of the type **gi_piestyle**. Its members define how the pieces of the pie chart are to be placed with respect to the other pieces. It has the following members:

```
PIS_ADJOIN      /* adjoining */
PIS_SEPARAT     /* separated */
```

apps is of the type **gi_chtapps**. It is a structure that specifies the visual attributes of the lines used to draw the elements of the pie chart itself, such as fill pattern and shading color. It contains the following members:

```
unsigned length;
gi_chtapp *values;
```

values is a pointer to an array of **gi_chtapp**. It is a structure that contains the following members:

```
gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;
dp_bool tranpare;
dp_color lncol;
```

joined is a Boolean value that specifies whether the elements of the pie chart (e.g., pie slices and text frames) are to merged as one with the pie chart, or if they are to remain separate graphic elements. If **joined** is **FALSE**, each graphics element will remain independent of the line chart and may be manipulated accordingly.

data is a pointer of the type **gi_chtdat**. Refer to **gi_adbacht()** for details.

wchild is a Boolean that, when set to **TRUE**, will cause a handle to the pie chart to be returned in **ret**. After which, graphic elements may be added to the handle. When set to **FALSE**, **ret** will contain a **NULL** value and the document editor will build the pie chart from the information contained in **gi_chtdat**. If a handle is returned, **gi_finishcht()** must be called to release it when done.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adpicht() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adbacht(), gi_finishcht()

gi_adpislce

NAME

gi_adpislce - add pie slice

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adpislce(h, box, props)
    gi_handle h;
    gi_box *box;                   /* NULL */
    gi_pislceprops *props;         /* NULL */
```

DESCRIPTION

The **gi_adpislce()** function is used to place a pie slice in a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_startgr()**, **gi_startbtn()**, **gi_startnbtn()**, **gi_startcluster()**, or **gi_adpicht()**.

The **box** argument is a pointer of the type **gi_box**. Refer to **gi_adcurve()** for a description of **gi_box**.

The **props** argument is a pointer to **gi_pislceprops**. It is a structure whose members define the appearance of the pie slice. **gi_pislceprops** contains the following members:

```
gi_brush brsh;
gi_shading shade;
gi_place center;
gi_place start;
gi_place stop;
dp_bool fixshape;
```

brsh is of the type **gi_brush**. It specifies the visual qualities of the lines used to draw the pie slice, such as solid or dashed lines, and their color. Refer to the description of **gi_startgframe()** for general information regarding **brsh**. The exception to the description of **brsh** in **gi_startgframe()** is with regards to the **stylebrush** member. The only two parameters that may be specified are **STB_INVISIBLE** and **STB_SOLID**.

shade is a structure of type **gi_shading**. It is used to define the appearance of the pie slice's interior. Its members are:

```
gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;
```

gray is of the type **gi_gray**, an enumerated variable that specifies the percentage of black, or saturation, to be used in making varying shades of the color gray. Refer to **gi_adbacht()** for a chart illustrating the available shades.

txrs is a structure of type **gi_textures**. It specifies the direction in which the texture is drawn in the pie slice or the type of texture that is to be placed in the pie slice. For example, textures may be drawn in a pie slice with a horizontal, vertical, or diagonal orientation. Also, a type of texture that may be placed in the pie slice is a polka dot pattern. **gi_textures** has the following members:

dp_bool vertical;
dp_bool horizontal;
dp_bool nwse;
dp_bool swne;
dp_bool polkadot;

Each variable is a Boolean value. The resulting texture will be the AND of the variables . That is, each variable that is set to **TRUE** will be placed as a texture in the graphic object.

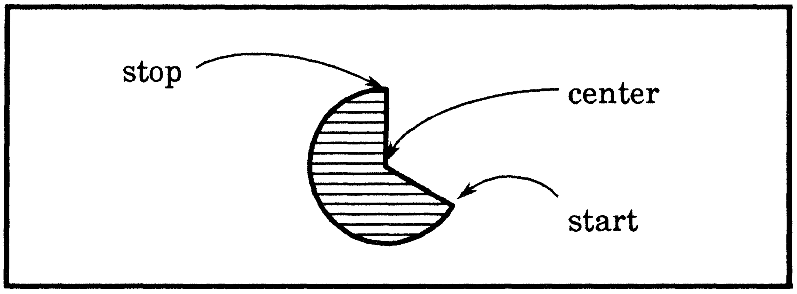
txrcol is a structure of type **dp_color**. Its members define the color that is to be used in drawing the texture, or foreground, of the pie slice's interior.

shdcol is a structure of type **dp_color**. Its members define the color to be used when drawing the background of the pie slice's interior.

center, **start**, and **stop** are values of the structure, **gi_place**. These values define the placement of the pie slice in **box**. The members of **gi_place** are:

int x;
int y;

x and **y** are integers that define an x and y axis location in **box**. Therefore, all grid locations are relative to **box.place**. **center** is the tip of the pie slice, or, if the pie were whole it could be considered the center of the pie. **start** and **stop** are the beginning and ending points on the edge, or circumference, of the pie slice. The arc of a pie slice goes from **start** to **stop** in a clockwise direction. **center**, **start**, and **stop** are all specified in units of micas. As shown below:



fixshape is a Boolean value that, when set to **TRUE**, indicates that the aspect ratio of a pie slice will remain intact when the user grows or shrinks it. A value of **FALSE** indicates that the aspect ratio of the pie slice will change freely. The value of this argument is always to be set to **TRUE** when adding pie slices.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adpislce() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adcurve(), gi_adbacht()

gi_adpoint

NAME

gi_adpoint - add point

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adpoint(h, box, props)
    gi_handle h;
    gi_box *box;                  /* NULL */
    gi_pointprops *props;         /* NULL */
```

DESCRIPTION

The **gi_adpoint()** function is used to add a point of a specific size and shape to a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_startgr()**, **gi_startbtn()**, **gi_startnbtn()**, **gi_startcluster()**, or **gi_adlncht()**.

The **box** argument is a pointer of the type **gi_box**. Refer to **gi_adcurve()** for a general description of **gi_box**. Note that the value of **box.dims** may be arbitrary because a point does not have dimensions, and so the value entered will be ignored.

The **props** argument is a pointer to **gi_pointprops**. It is a structure whose members define the appearance of the point. **gi_pointprops** contains the following members:

```
unsigned wth;
gi_ptstyle style;
gi_ptfill fill;
dp_color color;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

```
GSL W1          /* 1 width for Graphics Single Line */
GSL W2          /* 2 width for Graphics Single Line */
GSL W3          /* 3 width for Graphics Single Line */
GSL W4          /* 4 width for Graphics Single Line */
GSL W5          /* 5 width for Graphics Single Line */
GSL W6          /* 6 width for Graphics Single Line */
```

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

style is of the type **gi_ptstyle**. It is an enumerated variable that specifies the shape of the point. It may have one of the following values:

```
PTS_ROUND         /* round */
PTS_SQUARE        /* square */
PTS_TRIANGLE      /* triangle */
PTS_CROSS         /* cross */
PTS_INVISIBLE     /* invisible */
```

PTS_INVISIBLE may only be specified when placing a point in a line chart. This value is illegal in every other type of container.

fill is a structure of type **gi_ptfill**. It specifies if the point is to be drawn as a solid fill object or as an outline object with no fill. One of two values may be specified:**PTF_SOLID** or **PTF_HOLLOW**.

color is a structure of type **dp_color**. Its members are integers that specify a color that was obtained by a color extraction function, such as **dp_colfromname()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adpoint() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

dp_colfromname(), gi_adcurve()

gi_adrectangle

NAME

gi_adrectangle - add rectangle

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adrectangle(h, box, props)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_rectangleprops *props;       /* NULL */
```

DESCRIPTION

The **gi_adrectangle()** function is used to add a rectangle of a specific size and shape to a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_cluster()**, **gi_startnbtn()**, **gi_startbtn()**, **gi_adbacht()**, or **gi_adlncht()**.

The **box** argument is a pointer of type **gi_box**. It defines the size of the rectangle. Refer to **gi_adcurve()** for a description of **gi_box**.

The **props** argument is a pointer of the type **gi_rectangleprops**. It is a structure whose members define the appearance of the rectangle. Its members are:

```
gi_brush brsh;
gi_shading shade;
dp_bool fixshape;
```

brsh is a structure that defines the visual qualities of the lines used in tracing the border of the rectangle. It contains the following members:

```
unsigned wth;
gi_stlbrush stylebrush;
dp_color brushcolor;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

```
GSL W1          /* 1 width for Graphics Single Line */
GSL W2          /* 2 width for Graphics Single Line */
GSL W3          /* 3 width for Graphics Single Line */
GSL W4          /* 4 width for Graphics Single Line */
GSL W5          /* 5 width for Graphics Single Line */
GSL W6          /* 6 width for Graphics Single Line */
```

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

stylebrush defines how the lines are drawn, such as solid or dashed. It may have one of the following values:

```
STB_INVISIBLE      /* invisible */
STB_SOLID          /* solid */
STB_DASHED         /* dashed */
STB_DOTTED         /* dotted */
STB_DOUBLE         /* double */
STB_BROKEN         /* broken */
```

The value of **wth** is affected by the **stylebrush** specified. For example, the **wth** of **STB_DOUBLE** borders is 3 times the usual width because it consists of two lines separated by a gap equal to the width of the line.

brushcolor specifies the color to be used to display the lines that make up the edges of the graphic object. The value of color may be any color that is a member of **dp_color**.

shade is a structure of type **gi_shading**. It is used to define the appearance of the rectangle's interior. Its members are:

```
gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;
```

gray is of the type **gi_gray**, an enumerated variable that specifies the percentage of black, or saturation, to be used in making varying shades of the color gray. If **stylebrush** is set to **STB_INVISIBLE**, then **gray** may not be set to **GRY_NONE**, otherwise the rectangle will become invisible. Refer to **gi_adbacht()** for a chart illustrating the available shades.

txrs is a structure of type **gi_textures**. It specifies the direction in which the texture is drawn in the rectangle or the type of texture that is to be placed in the rectangle. For example, textures may be placed in an rectangle with a horizontal, vertical, or diagonal orientation. Also, a type of texture that may be placed in the rectangle is a polka dot pattern. **gi_textures** has the following members:

```
dp_bool vertical;
dp_bool horizontal;
dp_bool nwse;
dp_bool swne;
dp_bool polkadot;
```

Each variable is a Boolean value. The resulting texture will be the AND of the variables . That is, each variable that is set to **TRUE** will be placed as a texture in the graphic object.

txrcol is a structure of type **dp_color**. Its members define the color that is to be used in drawing the texture, or foreground, of the rectangle's interior.

shdcol is a structure of type **dp_color**. Its members define the color to be used when drawing the background in the rectangle's interior. This parameter is enabled only when the value of **gray** is **GRY_GRAY**. If the value of **gray** is any other value, **shdcol** is set to black{0, 0, 0}.

fixshape is a Boolean value that, when set to **TRUE**, indicates that the aspect ratio of the rectangle will remain intact when the user grows or shrinks it. A value of **FALSE** indicates that the aspect ratio of the rectangle will change freely.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adrectangle() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adcurve(), gi_adbacht()

gi__adtable

NAME

gi__adtable - add table

SYNOPSIS

```
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adtable(h, box, table, frprops, fixwidth, fixheight, wtcap, wbcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;                    /* NULL */
    di_ins table;                   /* NULL */
    gi_frameprops *frprops;         /* NULL */
    dp_bool fixwidth;               /* FALSE */
    dp_bool fixheight;              /* FALSE */
    dp_bool wtcap;                  /* FALSE */
    dp_bool wbcap;                  /* FALSE */
    dp_bool wlcap;                  /* FALSE */
    dp_bool wrcap;                  /* FALSE */
    ret_adtable *ret;               /* Returned */
```

DESCRIPTION

The **gi_adtable()** function is used to add a table frame into a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

Refer to the description of **box** in **gi_adffield()** for more information on **box**. Refer to **gi_startgframe()** for a description of the ***frprops**, and **w*cap** arguments.

The **table** argument is of the type **di_ins**. It is an opaque variable that contains the table handle that was returned by an earlier call to **ti_finishtable()**.

fixwidth and **fixheight** are Boolean values that indicate whether the width and/or height of a table frame is to remain static.

The **gi_adtable()** function sets the return information into the structure **ret_adtable**, which contains the following members:

```
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
```

The ***cap** arguments are each of the type **di_caption**, an opaque variable that contains a caption handle for the top, bottom, left, and right edges of the table frame, respectively. These handles may then be passed to various **di_ap*()** functions to append captions to the table.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adtable() will fail if one or more of the following are true:

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

ti_finishtable(), gi_adffield(), gi_startgframe()

gi_adtframe

NAME

gi_adtframe - add text frame

SYNOPSIS

```
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_adtframe(h, box, frprops, tfprops, wtext, wtcap, wbcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;                   /* NULL */
    gi_frameprops *frprops;        /* NULL */
    gi_tframeprops *tfprops;       /* NULL */
    dp_bool wtext;                 /* FALSE */
    dp_bool wtcap;                 /* FALSE */
    dp_bool wbcap;                 /* FALSE */
    dp_bool wlcap;                 /* FALSE */
    dp_bool wrcap;                 /* FALSE */
    ret_adtframe *ret;             /* Returned */
```

DESCRIPTION

The **gi_adtframe()** function is used to add a text frame to a specified graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, **gi_startcluster()**, **gi_adbacht()**, **gi_adlncht()**, **gi_adpicht()**.

Refer to the description of **box** in **gi_adffield** for more information on **box**. Refer to **gi_startgframe()** for a description of the ***frprops** and **w*cap** arguments. Refer to **gi_adffield()** for a description of **tfprops**.

The **wtext** argument is a Boolean value that specifies whether or not the frame is to have text. If a value of **TRUE** is specified, the **text** variable in the return value will be non-**NULL**. **DocIC** functions may then be used to add the text. Note the text must eventually be freed by a call to **di_reltext()**.

The **gi_adtframe()** function sets the return information into the structure **ret_adtframe**, which contains the following members:

```
di_text text;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
```

Refer to **gi_adffield()** for a description of ***cap**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adtframe() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adffield(), gi_startgframe(), di_reltext()

gi_adtriangle

NAME

gi_adtriangle - add triangle

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_adtriangle(h, box, props)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_triangleprops *props;        /* NULL */
```

DESCRIPTION

The **gi_adtriangle()** function is used to add a triangle of a specific size to a graphics container.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. Its two members, **place** and **dims**. specify the origin of the area in which the triangle will be placed and its size, relative to the graphics container. Refer to **gi_adcurve()** for a description of **gi_box**.

The **props** argument is a pointer to **gi_triangleprops**, a structure whose members define the appearance of the triangle. It contains the following members:

```
gi_brush brsh;
gi_shading shade;
gi_place p1;
gi_place p2;
gi_place p3;
dp_bool fixshape;
```

brsh is a structure that defines the visual qualities of the lines used in tracing the border of the triangle. It contains the following members:

```
unsigned wth;
gi_stlbrush stylebrush;
dp_color brushcolor;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

```
GSL W1          /* 1 width for Graphics Single Line */
GSL W2          /* 2 width for Graphics Single Line */
GSL W3          /* 3 width for Graphics Single Line */
GSL W4          /* 4 width for Graphics Single Line */
GSL W5          /* 5 width for Graphics Single Line */
GSL W6          /* 6 width for Graphics Single Line */
```

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

stylebrush defines how the lines are drawn, such as solid or dashed. It may have one of the following values:

```
STB_INVISIBLE        /* invisible */
STB_SOLID            /* solid */
STB_DASHED           /* dashed */
STB_DOTTED           /* dotted */
STB_DOUBLE           /* double */
STB_BROKEN           /* broken */
```

The value of **wth** is affected by the **stylebrush** specified. For example, the **wth** of **STB_DOUBLE** borders is 3 times the usual width because it consists of two lines separated by a gap equal to the width of the line.

brushcolor specifies the color to be used to display the lines that make up the edges of the graphic object. The value of color may be any color that is a member of **dp_color**.

shade is a structure of type **gi_shading**. It is used to define the appearance of the triangle's interior. Its members are:

```
gi_gray gray;
gi_textures txrs;
dp_color txrcol;
dp_color shdcol;
```

gray is of the type **gi_gray**, an enumerated variable that specifies the percentage of black, or saturation, to be used in making varying shades of the color gray. If **stylebrush** is set **STB_INVISIBLE**, then **gray** may not be set to **GRY_NONE**, otherwise the triangle will become invisible. Refer to **gi_adbacht()** for a chart illustrating the available shades.

txrs is a structure of type **gi_textures**. It specifies the direction in which the texture is drawn in the triangle or the type of texture that is to be placed in the triangle. For example, textures may be placed in an triangle with a horizontal, vertical, or diagonal orientation. Also, a type of texture that may be placed in the triangle is a polka dot pattern. **gi_textures** has the following members:

```
dp_bool vertical;
dp_bool horizontal;
dp_bool nwse;
dp_bool swne;
dp_bool polkadot;
```

Each variable is a Boolean value. The resulting texture will be the AND of the variables . That is, each variable that is set to **TRUE** will be placed as a texture in the graphic object.

txrcol is a structure of type **dp_color**. Its members define the color that is to be used in drawing the texture, or foreground, of the triangle's interior.

shdcol is a structure of type **dp_color**. Its members define the color to be used when drawing the background in the triangle's interior. This parameter is enabled only when the value of **gray** is **GRY_BLACK**. If the value of **gray** is any other value, **shdcol** is set to black {0, 0, 0}.

p1, **p2**, and **p3** are of the type **gi_place**. As mentioned in the description of **box**, **gi_place** is a structure that contains two integer members, **x** and **y**. When adding a triangle, these three members specify the

the x and y grid location for each of the three points of the triangle. **p1**, **p2**, and **p3** are specified in units of micas.

fixshape is a Boolean value that, when set to **TRUE**, indicates that the aspect ratio of a triangle will remain intact when the user grows or shrinks it. A value of **FALSE** indicates that the aspect ratio of the triangle will change freely.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_adtriangle() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_adcurve(), gi_adbacht()

gi_ap*btnprog

NAME

gi_apchartobtnprog, gi_apnparatobtnprog, gi_aptexttobtnprog - add to a CUSP button

SYNOPSIS

```
#include "DocICProps.h"
#include "GraphicsIC.h"
#include "XString.h"

int
gi_apchartobtnprog(to, char, foprops, num)
    gi_buttonprog to;
    XChar char;
    dp_fontprops *foprops;          /* NULL */
    unsigned num;                   /* 1 */

int
gi_apnparatobtnprog(to, paprops, foprops, num)
    gi_buttonprog to;
    dp_paraprops *paprops;          /* NULL */
    dp_fontprops *foprops;          /* NULL */
    unsigned num;                   /* 1 */

int
gi_aptexttobtnprog(to, text, foprops)
    gi_buttonprog to;
    XString text;
    dp_fontprops *foprops;          /* NULL */
```

DESCRIPTION

The following functions allow the user to add textual information to a CUSP button program.

gi_apchartobtnprog() is used to add a character to the button program.

gi_apnparatobtnprog() adds a new paragraph character with specified properties to the button program.

gi_aptexttobtnprog() adds a string with specified properties to button program.

For all three functions:

to is the button handle returned by an earlier call to **gi_startbtn()** or **gi_startnbtn()**.

char and **text** are the respective character and text strings to be inserted in the button program.

Refer to **dp_paraprops** and **dp_fontprops** in **dp_props** for a description of **foprops** and **paprops**.

num is the number of copies of the character or new paragraph characters to be added.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_ap*btnprog() will fail if one or more of the following are true:

Doc_DocumentFull No more room in the document.

Doc_ReadonlyDoc Document opened in ReadOnly mode.

Doc_OutOfDiskSpace Not enough disk space for the operation.

Doc_OutOfVM Not enough virtual memory for the operation.

Doc_BadParm One of the arguments specified is invalid.

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_startbtn(), gi_startnbtn()

gi_btnforaframe

NAME

gi_btnforaframe - button info for anchored frame

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"
#include "XString.h"

int
gi_btnforaframe(aframe, props, gridprops, ret)
    di_ins aframe;
    XString *props;
    gi_gridprops *gridprops;
    gi_buttonprog *ret;              /* Returned */
```

DESCRIPTION

The **gi_btnforaframe()** function is used to extract the properties of a button in an anchored CUSP button frame during enumeration. The button handle that is returned, **gi_buttonprog**, is a text object that points to CUSP programming code. It may be passed as an argument to **enumbtnprog()** to enumerate the text within the button.

The **aframe** argument is of the type **di_ins**, an enumerated variable that contains the handle of the frame in question. It was obtained by an earlier call to one of the **di_enumerate()** call-back procedures (**di_aframeproc()**).

The **props** argument is a pointer of the type XString. It is a return value in which the properties of a button are returned.

The **gridprops** argument is a pointer of the type **gi_gridprops**. It is a return value in which the grid properties of an anchored button are returned.

The **ret** argument is a pointer of the type **gi_buttonprog**, a handle to the button program object that contains the text contents of the anchored CUSP button.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_btnforaframe() will fail if one or more of the following are true:

Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumerate(), gi_enumbtnprog()

gi__enumbtnprog

NAME

gi__enumgbtnprog - enumerate button program

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_enumbtnprog(prog, procs, cdat. ret)
    gi_buttonprog prog;
    gi_btnenumprocs *procs;
    void *cdat;                    /* NULL */
    dp_bool *ret;                  /* Returned */
```

DESCRIPTION

The **gi_enumbtnprog()** function is used to enumerate the properties and text contents of a CUSP button.

prog is a variable of the type **gi_buttonprog**. Refer to **gi_startnbtn()** for a description of **gi_buttonprog**.

procs is a pointer of the type **gi_btnenumprocs**, a user-supplied structure containing the user's call-back procedures. **gi_btnenumprocs** contains the following members:

```
di_newparaproc *newpara;
di_textproc *text;
```

newpara is a pointer of the type **di_newparaproc**, a call-back procedure that is called when a new paragraph character is encountered in the text.

text is a pointer of the type **di_textproc**, a call-back procedure that is called whenever a substring of text is encountered. The whole substring is passed as a parameter. Therefore, **di_textproc** may be called repeatedly, once for each substring of text having the same properties.

cdat is passed to each call-back procedure during enumeration.

ret will be true if **gi_enumbtnprog()** encounters an object it does not recognize, or an object for which a call-back procedure was not supplied.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_enumbtnprog() will fail if one or more of the following are true:

Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_btnforaframe(), gi_enumerate()

gi_enumerate

gi_enumerate - reading graphics

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"

int
gi_enumerate(gcont, procs, cdat. ret)
    di_ins gcont;
    gi_enumprocs *procs;
    void *cdat;                      /* NULL */
    dp_bool *ret;                    /* Returned */
```

CALLBACK PROCEDURE

```
dp_bool
gi_bachtproc(cdat, box, props, data, chart)
    void *cdat;
    gi_box *box;
    gi_bachtprops *props;
    gi_chtdat *data;
    di_ins chart;

dp_bool
gi_bmproc(cdat, box, bmprops, frprops)
    void *cdat;
    gi_box *box;
    gi_bmprops *bmprops;
    gi_frameprops *frprops;

dp_bool
gi_buttonproc(cdat, gcont, box, name, gridprops, frprops, prog)
    void *cdat;
    di_ins gcont;
    gi_box *box;
    XString name;
    gi_gridprops *gridprops;
    gi_frameprops *frprops;
    gi_buttonprog prog;

dp_bool
gi_clusterproc(cdat, gcont, box)
    void *cdat;
    di_ins gcont;
    gi_box *box;

dp_bool
gi_curveproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_curveprops *props;
```

```
dp_bool
gi_ellipseproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_ellipseprops *props;

dp_bool
gi_tfieldproc(cdat, box, fiprops, frprops, tfprops, paprops, foprops, cont)
    void *cdat;
    gi_box *box;
    dp_fldprops *fiprops;
    gi_frameprops *frprops;
    gi_tframeprops *tfprops;
    dp_paraprops *paprops;
    dp_fontprops *foprops;
    di_field cont;

dp_bool
gi_frameproc(cdat, gcont, box, frprops, gfprops)
    void *cdat;
    di_ins gcont;
    gi_box *box;
    gi_frameprops *frprops;
    gi_gframeprops *gfprops;

dp_bool
gi_lnchtproc(cdat, box, props, data, chart)
    void *cdat;
    gi_box *box;
    gi_lnchtprops *props;
    gi_chtdat *data;
    di_ins chart;

dp_bool
gi_lineproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_lineprops *props;

dp_bool
gi_pichtproc(cdat, box, props, data, chart)
    void *cdat;
    gi_box *box;
    gi_pichtprops *props;
    gi_chtdat *data;
    di_ins chart;

dp_bool
gi_pislceproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_pislceprops *props;
```

```
dp_bool
gi_pointproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_pointprops *props;

dp_bool
gi_rectangleproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_rectangleprops *props;

dp_bool
gi_tableproc(cdat, box, table, frprops, fixwidth, fixheight)
    void *cdat;
    gi_box *box;
    di_ins table;
    gi_frameprops *frprops;
    dp_bool fixwidth;                /* FALSE */
    dp_bool fixheight;               /* FALSE */

dp_bool
gi_tframeproc(cdat, box, frprops, tfprops, cont)
    void *cdat;
    gi_box *box;
    gi_frameprops *frprops;
    gi_tframeprops *tfprops;
    di_text cont;

dp_bool gi_triangleproc(cdat, box, props)
    void *cdat;
    gi_box *box;
    gi_triangleprops *props;
```

DESCRIPTION

The **gi_enumerate**() function is used to read the contents of a graphics frame. It takes a graphics container handle, a list of call-back procedures, and user data as arguments. Typically, a call-back procedure is supplied for each type of graphic object that is in the graphics container. Once called, **gi_enumerate()** proceeds through each container, calling the appropriate procedure for each type of object encountered.

Each call-back procedure takes arguments that describe the properties of the object in question. These properties are temporary, and will be invalidated upon completion of the procedure call. If you want to save these properties, you must explicitly copy them.

gi_rel* functions should not be called by any of the **gi_enumerate()** call-back procedures because the handles for each **gi_rel*** function is automatically released once it has been processed.

In the case of a CUSP button, a cluster, or a nested graphics frame, **gi_enumerate()** may be called recursively to extract the contents of nested frames.

gi_enumprocs contains the following members:

```
gi_bachtproc *bacht;
gi_bmproc *bm;
gi_buttonproc *button;
gi_clusterproc *cluster;
gi_curveproc *curve;
gi_ellipseproc *ellipse;
gi_ffieldproc *ffield;
gi_frameproc *frame;
gi_lnchtproc *lncht;
gi_lineproc *line;
gi_pichtproc *picht;
gi_pislceproc *pislce;
gi_pointproc *point;
gi_rectangleproc *rectangle;
gi_tableproc *table;
gi_tframeproc *tframe;
gi_triangleproc *triangle;
```

Related enumeration functions are **di_enumerate()** and **gi_enumbtnprog()**. They are used to enumerate the contents of a document or text container and a CUSP button, respectively.

ret will be **TRUE** if **gi_enumerate()** encounters an object it does not recognize, or an object for which a callback procedure was not supplied.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_enumerate() will fail if one or more of the following are true:

Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumerate(), gi_enumbtnprog()

gi_finish*

NAME

gi_finishnbtn, gi_finishcluster, gi_finishgr, gi_finishframe, gi_finishcht - finish routine

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"

int
gi_finishcht(chart)
   gi_handle chart;

int
gi_finishcluster(ch)
   gi_handle ch;

int
gi_finishgframe(gfh)
   gi_handle gfh;

int
gi_finishgr(h, ret)
   gi_handle h;
   di_ins *ret;                    /* Returned */

int
gi_finishnbtn(bfh)
   gi_handle bfh;
```

DESCRIPTION

The **gi_finish*()** functions are used to signal that no more objects are to be added to the respective graphics container. Calling a **gi_finish*()** function will free up the respective handle.

bfh, **ch**, **h**, and **gfh** arguments are the handles obtained from corresponding **gi_start*()** functions. The **chart** argument is obtained from **gi_adbacht()**, **gi_adlncht()** or **gi_adpicht()** functions when **wchild** is set to **TRUE.**

gi_finishgr() returns **di_ins**. Typically, **di_ins** is passed as an argument to **di_apaframe().**

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_finish() will fail if one or more of the following are true:

Doc_IllegalHandle The specified handle is illegal.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_start*(), di_apaframe(), gi_adbacht(), gi_adlncht(), gi_adpicht()

gi_getgframeprops

NAME

gi_getgframeprops - get graphics frame props

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"

int
gi_getgframeprops(aframe, ret)
    di_ins aframe;
    gi_gframeprops *ret;                /* Returned */
```

DESCRIPTION

The **gi_getgframeprops()** function is used to retrieve the name, description, and grid properties of an anchored graphics frame.

The **aframe** argument is of the type **di_ins**, an enumerated variable that contains the handle of the anchored frame in question.

The requested property values are stored in **ret**. It is a structure that contains the following members:

```
XString name;
XString desc;
gi_gridprops gridprops;
```

name is name of an anchored graphics frame. **desc** is description of an anchored graphics frame. Refer **gi_setgframeprops()** for a description of **gi_gridprops**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_getgframeprops() will fail if one or more of the following are true:

Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_enumerate()

gi__get*def

NAME

gi__get* - get default properties

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_getbachtpropsdef(props)
    gi_bachtprops *props;           /* Returned */

int
gi_getbmdispdef(disp)
    gi_bmdisp *disp;                /* Returned */

int
gi_getbmpropsdef(props)
    gi_bmprops *props;              /* Returned */

int
gi_getbmscalpropsdef(props)
    gi_bmscalprops *props;          /* Returned */

int
gi_getboxdef(box)
    gi_box *box;                    /* Returned */

int
gi_getchtappdef(app)
    gi_chtapp *app;                 /* Returned */

int
gi_getchtdatdef(dat)
    gi_chtdat *dat;                 /* Returned */

int
gi_getcurvepropsdef(props)
    gi_curveprops *props;           /* Returned */

int
gi_getellipsepropsdef(props)
    gi_ellipseprops *props;         /* Returned */

gi_getframepropsdef(props)
    gi_frameprops *props;           /* Returned */

int
gi_getgframepropsdef(props)
    gi_gframeprops *props;          /* Returned */

int
gi_getgridpropsdef(props)
    gi_gridprops *props;            /* Returned */
```

```
int
gi_getlinepropsdef(props)
    gi_lineprops *props;            /* Returned */

int
gi_getlnchtappdef(app)
    gi_lnchtapp *app;               /* Returned */

int
gi_getlnchtpropsdef(props)
    gi_lnchtprops *props;           /* Returned */

int
gi_getpichtpropsdef(props)
    gi_pichtprops *props;           /* Returned */

int
gi_getpislcepropsdef(props)
    gi_pislceprops *props;          /* Returned */

int
gi_getpointpropsdef(props)
    gi_pointprops *props;           /* Returned */

int
gi_getrectanglepropsdef(props)
    gi_rectangleprops *props;       /* Returned */

int
gi_gettframepropsdef(props)
    gi_tframeprops *props;          /* Returned */

int
gi_gettrianglepropsdef(props)
    gi_triangleprops *props;        /* Returned */
```

DESCRIPTION

The following functions all return the default values of their respective properties. The properties that are returned may be modified individually and then passed in a call to a suitable function to set the properties of an object. The actual default values for the properties are shown as C comments.

The **gi_getbachtpropsdef()** function is used to return the default bar chart properties.

```
double units;                    /* 1.0 */
unsigned div;                    /* 0 */
gi_barscale scale;               /* BS_STICK (single tick) */
dp_color sclcol;                 /* 0, 0, 0 */
gi_balayout layout;              /* BL_STACKED */
gi_baspacing spacing;            /* BSP_HALF (half spacing) */
gi_baorient orient;              /* BO_VER (vertical) */
dp_bool key;                     /* FALSE */
dp_bool bafloat;                 /* FALSE */
dp_bool mirror;                  /* FALSE */
gi_chtapps *apps;                /* NULL */
dp_bool joined;                  /* TRUE */
```

The **gi_getbmdispdef()** function is used to return the default bitmap display properties.

```
enum {
    BM_INTERNAL,
    BM_FILE
    } type;                      /* BM_FILE */
union {
    gi_bmdat *bm;
    XString name;                /* NULL */
    } u;
```

The **gi_getbmpropsdef()** function is used to return the default bitmap properties.

```
int xoffset;                     /* 0 */
int yoffset;                     /* 0 */
XString prntfile;                /* NULL */
gi_bmdisp dispsou;               /* see gi_bmdispdef() */
gi_bmscalprops scalprops;        /* see gi_bmscalpropsdef() */
dp_bool remotefile;              /* FALSE */
dp_color bitcol;                 /* CL_BLACK */
```

The **gi_getbmscalpropsdef()** function is used to return the default bitmap scale properties.

```
enum {
    BMS_PRNTRES,
    BMS_FIXED,
    BMS_AUTOMATIC,
    } type;                      /* BMS_AUTOMATIC */
union {
    unsigned res;
    gi_scalfix  fixed;
    enum {
        SHP_SIMILAR,
        SHP_FILLUP
        } shape;                 /* SHP_SIMILAR */
    } u;
```

The **gi_getboxdef()** function is used to return the default box properties.

```
gi_place place;                  /* 1000, 1000 */
gi_dims dims;                    /* 1000, 1000 */
```

The **gi_getchtappdef()** function is used to return the default chart appearances properties.

```
gi_gray gray;                  /* GRY_NONE */
gi_textures txrs;              /* all FALSE */
dp_color txrcol;               /* 0, 0, 0 */
dp_color shdcol;               /* 0, 0, 0 */
dp_color lncol;                /* 0, 0, 0 */
```

The **gi_getchtdatdef()** function is used to return the default chart data properties.

```
XString title;                 /* NULL */
gi_dataset datset;             /* DAS_COLUMN */
dp_lang lang;                  /* LANG_USE (USEnglish) */
gi_datsource datsou;           /* DTS_PS, TFO_BYCOL */
gi_labels *collabl;            /* NULL */
gi_labels *rowlabl;            /* NULL */
gi_datvalues *values;          /* NULL */
```

The **gi_getcurvepropsdef()** function is used to return the default curve properties.

```
gi_brush brsh;                 /* 71, STB_SOLID, 0, 0, 0 */
gi_lnend lnenw;                /* LE_SQUARE */
gi_lnend lnese;                /* LE_SQUARE */
gi_lnedhd lnhnw;               /* LEH_NONE */
gi_lnedhd lnhse;               /* LEH_NONE */
gi_place plnw;                 /* 1000, 0 */
gi_place plapx;                /* 0, 0 */
gi_place plse;                 /* 0, 1000 */
gi_place plpek;                /* 0, 0 */
dp_bool eccentric;             /* TRUE */
unsigned eccentricity;         /* 32768 */
dp_bool fixangle;              /* FALSE */
```

The **gi_getellipsepropsdef()** function is used to return the default ellipse properties.

```
gi_brush brsh;                 /* 71, STB_SOLID, {0, 0, 0} */
gi_shading shade;              /* GRY_NONE, all FALSE, {0, 0, 0}, {10000, 0, 0} */
dp_bool fixshape;              /* FALSE */
```

The **gi_getframepropsdef()** function is used to return the default frame properties.

```
gi_brush brsh;                 /* 71, STB_SOLID, {0, 0, 0} */
dp_bool fixshape;              /* FALSE */
unsigned mgns[4];              /* 0, 0, 0, 0 */
di_caption capcont[4];         /* NULL, NULL, NULL, NULL */
dp_color bgcol;                /* 10000, 0, 0 */
dp_bool tranpare;              /* FALSE */
```

The **gi_getgframepropsdef()** function is used to return the default graphics frame properties.

```
XString name;                  /* NULL */
XString desc;                  /* NULL */
gi_gridprops gridprops;        /* see gi_bmscalpropsdef() */
```

The **gi_getgridpropsdef()** is used to return the default grid properties.

```
dp_bool act;                    /* FALSE */
gi_gridstyle style;             /* GRD_DOT */
gi_gridsize size;               /* GRD_8P */
gi_place offset;                /* 0, 0 */
```

The **gi_getlinepropsdef()** function is used to return the default line properties.

```
gi_brush brsh;                  /* 71, STB_SOLID, {0, 0, 0} */
gi_lnend lnenw;                 /* LE_SQUARE */
gi_lnend lnese;                 /* LE_SQUARE */
gi_lnedhd lnhnw;                /* LEH_NONE */
gi_lnedhd lnhse;                /* LEH_NONE */
gi_lndirct dirct;               /* LD_WE */
dp_bool fixangle;               /* FALSE */
```

The **gi_getlnchtappdef()** function is used to return the default line chart appearances properties.

```
unsigned psize;                 /* 3 */
gi_ptfill pfill;                /* PTF_SOLID */
gi_ptstyle pstyle;              /* PTS_ROUND */
dp_color pcolor;                /* 0, 0, 0 */
gi_curvetype ctype;             /* CUT_STRAIGHT */
gi_brush cbrush;                /* 71, STB_SOLID, {0, 0, 0} */
```

The **gi_getlnchtpropsdef()** function is used to return the default line chart properties.

```
double xunits;                  /* 1.0 */
double yunits;                  /* 1.0 */
double xmax;                    /* 0.0 */
double xmin;                    /* 0.0 */
double ymax;                    /* 0.0 */
double ymin;                    /* 0.0 */
unsigned xdiv;                  /* 0 */
unsigned ydiv;                  /* 0 */
gi_axtype xaxtype;              /* AXT_STICK (single tick) */
gi_axtype yaxtype;              /* AXT_STICK (single tick) */
gi_rotation axorient;           /* RT_NORMAL */
dp_bool key;                    /* FALSE */
dp_color scalcol;               /* 0, 0, 0 */
gi_lnchtapps *apps;             /* NULL */
dp_bool joined;                 /* TRUE */
```

The **gi_getpichtpropsdef()** function is used to return the default pie chart properties.

```
unsigned wth;                   /* 71 */
gi_piestyle style;              /* PIS_ADJOIN */
gi_chtapps *apps;               /* NULL */
dp_bool joined;                 /* TRUE */
```

The **gi_getpislcepropsdef()** function is used to return the default pie slice properties.

```
gi_brush brsh;          /* 71, STB_SOLID, {0, 0, 0} */
gi_shading shade;       /* GRY_NONE, all FALSE, {0, 0, 0}, {0, 0, 0} */
gi_place center;        /* 500, 500 */
gi_place start;         /* 500, 0 */
gi_place stop;          /* 0, 500 */
dp_bool fixshape;       /* FALSE */
```

The **gi_getpointpropsdef()** function is used to return the default point properties.

```
unsigned wth;           /* 71 */
gi_ptstyle style;       /* PTS_ROUND */
gi_ptfill fill;         /* PTF_SOLID */
dp_color color;         /* 0, 0, 0 */
```

The **gi_getrectanglepropsdef()** function is used to return the default rectangle properties.

```
gi_brush brsh;          /* 71, STB_SOLID, {0, 0, 0} */
gi_shading shade;       /* GRY_NONE, all FALSE, {0, 0, 0}, {0, 0, 0} */
dp_bool fixshape;       /* FALSE */
```

The **gi_gettframepropsdef()** function is used to return the default text frame properties.

```
dp_bool expr;           /* FALSE */
dp_bool expb;           /* FALSE */
dp_tframeprops props;   /* see dp_tframepropsdef() */
```

The **gi_gettrianglepropsdef()** function is used to return the default triangle properties.

```
gi_brush brsh;          /* 71, STB_SOLID, {0, 0, 0} */
gi_shading shade;       /* GRY_NONE, all FALSE, {0, 0, 0}, {0, 0, 0} */
gi_place p1;            /* 500, 0 */
gi_place p2;            /* 0, 1000 */
gi_place p3;            /* 1000, 1000 */
dp_bool fixshape;       /* FALSE */
```

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_get*def() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.

gi_relbtnprog

NAME

gi_relbtnprog - release button program

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_relbtnprog(btnprog)
    gi_buttonprog *btnprog;
```

DESCRIPTION

The **gi_relbtnprog()** function is used to release handles obtained by calls to **gi_startbtn()** or **gi_startnbtn()**.

The **btnprog** argument is an opaque variable that points to the handle of the button program to be freed. A call to this function will set the respective handle to **NULL**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_relbtnprog() will fail if one or more of the following are true:

Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_startbtn(), gi_startnbtn()

gi_setgframeprops

NAME

gi_setgframeprops - set graphics frame properties

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"

int
gi_setgframeprops(aframe, props)
    di_ins aframe;
    gi_gframeprops *props;            /* NULL */
```

DESCRIPTION

The **gi_setgframeprops()** function is used to set the properties of a graphics frame.

The **aframe** argument is an unsigned opaque variable that contains the frame handle returned by an earlier call to **di_apaframe()**.

The **props** argument is a pointer of the type **gi_gframeprops**. It is a structure that contains specific frame properties. **gi_gframeprops** contains the following members:

```
XString name;
XString desc;
gi_gridprops gridprops;
```

name and **desc** are the name and description of the graphics frame for which the properties are to be set.

gi_gridprops is a structure that defines the composition of the grid. It contains the following members:

```
dp_bool act;
gi_gridstyle style;
gi_gridsize size;
gi_place offset;
```

act, short for activity, indicates the state of the grid. When **act** is **TRUE**, the grid is displayed in the graphics frame. **style** and **size** describe the respective grid type and the interval between grid marks .

style may have one of the following values:

```
GRD_DOT           /* dot */
GRD_PLUS          /* plus */
GRD_TICK          /* tick */
```

size is specified in units of points, where there are 72 points per inch. **size** may have one of the following values:

```
GRD_4P          /* 4 point */
GRD_8P          /* 8 point */
GRD_12P         /* 12 point */
GRD_16P         /* 16 point */
GRD_32P         /* 32 point */
```

offset describes the shift values of the upper left grid point relative to the upper left corner of the graphics frame. **offset** is of the type **gi_place**. It is a structure whose two members are integers that define the x and y grid locations. Setting both members to 0 indicates that no offset is desired. **offset** is specified in units of points, where 72 points are the equivalent of one inch.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_setgframeprops() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_apaframe()

gi_startbtn, gi_finishbtn

NAME

gi_startbtn, gi_finishbtn - create and complete an anchored CUSP button frame

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"
#include "XString.h"

int
gi_startbtn(doc, name, gridprops, wprog, ret)
    di_doc doc;
    XString name;                      /* NULL */
    gi_gridprops *gridprops;           /* NULL */
    dp_bool wprog;                     /* FALSE */
    ret_startbtn *ret;                 /* Returned */

int
gi_finishbtn(h, ret)
    gi_handle h;
    di_ins *ret;                       /* Returned */
```

DESCRIPTION

The **gi_startbtn()** function is used to begin the creation of an anchored CUSP button.

The **doc** argument is of the type **di_doc**, an enumerated variable that contains the document handle returned by an earlier call to **di_start()** or **di_startap()**.

The **name** argument is an XString variable that contains either a valid button name or **NULL**. If **name** is set to **NULL**, **gi_startnbtn()** will generate a new unique name for the button, such as Button1, Button2, etc.

The **gridprops** argument is a pointer of the type **gi_gridprops**. It is a structure whose members define the style, size, and offset of the grid to be used in the new anchored button. Refer **gi_setgframeprops()** for a description of **gi_gridprops**.

The **wprog** argument is a Boolean value that determines whether the returned **gi_buttonprog** will be valid or NULL. Pass **FALSE** as the value of this argument if you do not intend to use **gi_ap*tobtnprog()** functions to append data to the button during the current programming session. Pass **TRUE** as the value of this argument to get a non-**NULL** program for this button. Complete the implementation of the resulting button, **gi_buttonprog**, by calling the various **gi_ap*tobtnprog()** functions. If **wprog** is set to **TRUE**, **gi_relbtnprog()** must be called to release **gi_buttonprog** after all the desired data has been appended and before calling **gi_finishbtn()**. **gi_finishbtn()** finishes all the non-program aspects of button creation and returns an instance to pass as the **cont** parameter of **di_apaframe()**.

The **ret** argument is a pointer of the type **ret_startbtn**. It is a structure in which will be placed the return information. **ret_startbtn** contains the following members:

```
gi_handle h;
gi_buttonprog prog;
```

h is the graphic handle of a CUSP button.

prog is a pointer of the type **gi_buttonprog**. It is an enumerated variable that contains the button program data and is supplied as an argument to **gi_ad*btn()** functions.

The **gi_finishbtn()** function is used to terminate the creation of an anchored button. This function is called after all the desired data has been added to the anchored button.

The **h** argument to **gi_finishbtn()** is of the type **gi_handle**. It is an enumerated variable that contains the anchored button handle returned by an earlier call to **gi_startbtn()**.

ret is a pointer of the type **di_ins**, an enumerated variable that is usually passed as an argument to **gi_apaframe()** only.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_startbtn() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

gi_finishbtn() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_relbtnprog(), gi_finishbtn(), di_apaframe(), gi_setgframeprops()

gi_startcluster

NAME

gi_startcluster - start cluster

SYNOPSIS

```
#include "GraphicsIC.h"

int
gi_startcluster(h, box, ret)
   gi_handle h;
   gi_box *box;                    /* NULL */
   gi_handle *ret;                 /* Returned */
```

DESCRIPTION

The **gi_startcluster()** function is used to create a set of graphic objects. Graphic objects may then be placed at the location specified in the **box** argument by passing the handle returned by this function, **gi_handle**, to the appropriate **gi_ad*()** functions.

The **h** argument is the graphics handle returned by an earlier call to **gi_startgr()**, **gi_startgframe()**, **startgframe()**, or **gi_startbtn()**.

The **box** argument is a pointer of the type **gi_box**. It's two members, **place** and **dims** specify the location of the container in which is to be placed graphic objects relative to the anchored frame, as well as the size of the container.

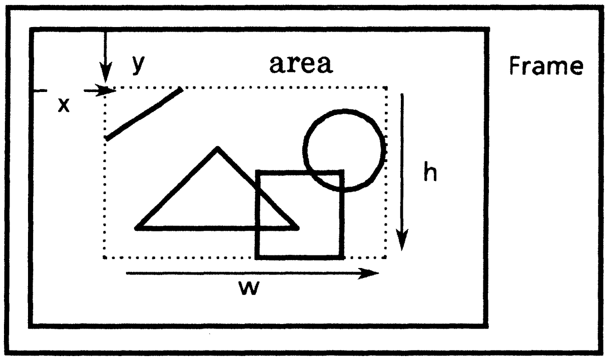
```
gi_place place;
gi_dims dims;
```

gi_place contains two integer variables **x** and **y**. These two variables indicate the grid location of the box origin. **gi_dims** contains two integer variables **w** and **h**. These two variables indicate the width and height of the box with respect to the box origin. Both **place** and **dims** are specified in units of micas.

A {0, 0} grid location indicates the upper left corner of a frame. Increasing the value of **x** causes the placement location to shift towards the right. Increasing the value of **y** causes the placement location to shift downwards. It is illegal to specify negative **w** and **h** values, therefore an object's **dims.place** must always correspond to the upper left corner of a box. It is legal to specify negative **x** and **y** values.

box.dims defines the area in which may be placed graphic objects. Increasing the value of **w** causes the frame to grow towards the right. Increasing the value of **h** causes the frame to grow in a downward direction. If an attempt is made to fit a graphic object within a frame that is too small to accommodate the graphic object, via calls to **gi_ad***() functions, only that portion of the object which lies inside the frame will be displayed. Those portions of the object which lie outside the frame still exist but are not displayed.

For example, to define the location and area to be occupied by a cluster of graphic objects, **box.dims** and **box.place** would have the following effect:



The cluster will be placed in the resulting area defined by **w** and **h**, relative to the location specified by **x** and **y**. Once the cluster has been defined, graphic objects may be placed within it.

Once all the desired graphic objects have been added to the cluster, call **gi_finishcluster()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_startcluster() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_finishcluster()

gi__startgframe

NAME

gi__startgframe - start graphics frame

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"

int
gi_startgframe(h, box, frprops, gfprops, wtcap, wbcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;                    /* NULL */
    gi_frameprops *frprops;         /* NULL */
    gi_gframeprops *gfprops;        /* NULL */
    dp_bool wtcap;                  /* FALSE */
    dp_bool wbcap;                  /* FALSE */
    dp_bool wlcap;                  /* FALSE */
    dp_bool wrcap;                  /* FALSE */
    ret_startgframe *ret;           /* Returned */
```

DESCRIPTION

The **gi_startgframe()** function is used to nest a graphics frame within a graphics container. The resulting frame will have a set of user-defined properties. The handle returned by this function may then be passed as an argument to other **gi_ad*()** functions.

The **h** argument is the graphics container handle representing the container into which the nested graphics frame is to be placed. This handle may come from several places, most notably **gi_startgr()**, **gi_startgframe()**, **gi_startbtn()**, **gi_startnbtn()**, or **gi_startcluster()**.

The **box** argument is a pointer of the type **gi_box**. Refer to **gi_startcluster()** for a description of **gi_box**.

The **frprops** argument is a pointer of the type **gi_frameprops**. It is a structure that defines the common properties of the graphics frame. **gi_frameprops** contains the following members:

```
gi_brush brsh;
dp_bool fixshape;
unsigned mgns[4];
di_caption capcont[4];
dp_color bgcol;
dp_bool tranpare;
```

brsh is a structure that defines the visual qualities of the lines comprising the edges of the frame. It contains the following variables:

```
unsigned wth;
gi_stlbrush stylebrush;
dp_color brushcolor;
```

wth is the width of lines, specified in units of micas. The standard brush widths may have one of the following value:

GSL W1	/* 1 width for Graphics Single Line */
GSL W2	/* 2 width for Graphics Single Line */
GSL W3	/* 3 width for Graphics Single Line */
GSL W4	/* 4 width for Graphics Single Line */
GSL W5	/* 5 width for Graphics Single Line */
GSL W6	/* 6 width for Graphics Single Line */

Each value corresponds to 35, 71, 106, 141, 176, and 212 micas, respectively. Non-standard brush widths will result in an error.

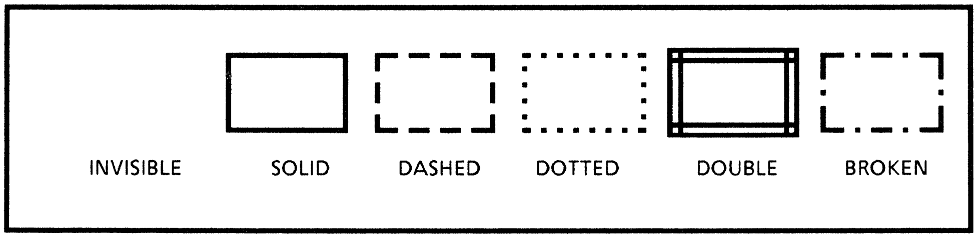
stylebrush defines how the lines are to be drawn, such as solid or dashed. It may have one of the following values:

STB_INVISIBLE	/* invisible */
STB_SOLID	/* solid */
STB_DASHED	/* dashed */
STB_DOTTED	/* dotted */
STB_DOUBLE	/* double */
STB_BROKEN	/* broken */

The **wth** of **STB_DOUBLE** borders is 3 times the usual width because it consists of two lines separated by a gap equal to the width of the line. In this case, the brush widths may have one of the following values:

GDL W1	/* 1 width for Graphics Double Line */
GDL W2	/* 2 width for Graphics Double Line */
GDL W3	/* 3 width for Graphics Double Line */
GDL W4	/* 4 width for Graphics Double Line */
GDL W5	/* 5 width for Graphics Double Line */
GDL W6	/* 6 width for Graphics Double Line */

Each value corresponds to 106, 212, 318, 423, 529, and 635 micas, respectively. The following are examples of brush styles:



brushcolor specifies the color to be used to display the lines that make up the edges of the graphics frame. The value of color may be any color that is a member of **dp_color**.

fixshape is a Boolean value that, when set to **TRUE**, indicates that the aspect ratio of a frame will remain intact when the user grows or shrinks the **box** that contains it. A value of **FALSE** indicates that the aspect ratio of the graphic object will change in proportion to the changes made to the **box** that contains it.

mgns is an array used to define the margins outside the frame. It requires four values. The values set the top, bottom, left, and right margins, respectively.

capcont is an array used to specify the captions associated with the frame. Its four elements are opaque caption handles. The **capcont** parameter is only meaningful during enumeration and when passing the caption handles as arguments to suitable **gi_ad*()** functions, and not when calling **gi_start*()**, since the contents of each caption is added after the frame is created.

bgcol is a structure comprised of integers that define the background color of the frame. These integers are returned from a previous call to a color translation function, such as **dp_colfromname()**.

tranpare is a Boolean value that specifies if the background color of the frame is to be white or transparent when the value of **bgcol** is {0,0,0} (i.e., white). A value of **TRUE** indicates that the background is to be transparent when the color is {0,0,0}. A value of **FALSE** indicates that the background is to be a solid white color.

gfprops is a pointer of the type **gi_gframeprops**. It is a structure whose members are used to set specific frame properties. Refer to the description of **gi_setgframeprops()** for more information.

w*cap arguments are Boolean values that specify whether or not the frame is to have captions. If a value of **TRUE** is specified for a **w*cap** argument. the respective ***cap** return value will be non-**NULL**. These caption arguments are used to set the top, bottom, left, and right captions, respectively. **DocIC** functions may then be used to add text to each caption. Note that the caption must eventually be freed by a call to **di_relcap()**.

gi_startgframe() sets **ret** as the return information. **ret_startgframe** contains the following members:

```
gi_handle gfh;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
```

gfh is the handle to the newly created graphics frame. **tcap**, **bcap**, **lcap** and **rcap** are the frame captions for top, bottom, left and right, respectively.

Once the desired graphic objects have been added to the frame, **gi_finishgframe()** must be called to release the handle and resources allocated by the system.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_startgframe() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_relcap(), dp_colfromname(), gi_finishgframe(), gi_setgframeprops()

gi_startgr

NAME

gi_startgr - start to create an anchored frame

SYNOPSIS

```
#include "DocIC.h"
#include "GraphicsIC.h"

int
gi_startgr(doc, ret)
    di_doc doc;
    gi_handle *ret;                /* Returned */
```

DESCRIPTION

The **gi_startgr()** function is used to create a graphics frame in a document so that graphic objects may be placed within the resulting frame. This function returns a graphics handle, **gi_handle**, which is an opaque variable that contains a graphics container. A graphics container is simply an object that may contain graphic objects. A graphics container may be a nested graphics frame, a CUSP button within a graphics frame, or a similar construct, such as a chart.

The **doc** argument is the document file handle that was returned by an earlier call to either **di_start()** or **di_startap()**.

Once all the desired objects have been added to a frame, call **gi_finishgr()** to free the handle and the resources allocated to that graphics frame.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_startgr() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

di_startap(), gi_startgframe(), gi_startnbtn()

gi_startnbtn

NAME

gi_startnbtn - start nested button

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "GraphicsIC.h"
#include "XString.h"

int
gi_startnbtn(h, box, name, gridprops, frprops, wprog, wtcap, wlcap, wrcap, ret)
    gi_handle h;
    gi_box *box;
    XString name;
    gi_gridprops *gridprops;
    gi_frameprops *frprops;
    dp_bool wprog;               /* FALSE */
    dp_bool wtcap;               /* FALSE */
    dp_bool wbcap;               /* FALSE */
    dp_bool wlcap;               /* FALSE */
    dp_bool wrcap;               /* FALSE */
    ret_startnbtn *ret;          /* Returned */
```

DESCRIPTION

The **gi_startnbtn()** function is used to create a CUSP button in a frame. The resulting button may then have CUSP code placed inside it via the **prog** argument to **ret_startnbtn**.

The **h** argument is the graphics container handle returned by an earlier call to **gi_startgframe()**, **gi_cluster()**, **gi_startnbtn()**, or **gi_startbtn()**.

The **box** argument is a pointer of the type **gi_box**. Refer to **gi_adffield()** for a description of **gi_box**.

The **name** argument is the default name of the button. If this parameter is left **NULL**, **gi_startnbtn()** will generate a new unique name for the button, such as Button1, Button2, etc.

gridprops is a pointer of type **gi_gridprops**. It is a structure whose members determine the composition of the grid. Refer to the description of **gi_setgframeprops()** for more information.

frprops is a pointer of type **gi_frameprops**. It is a structure whose members determine the common properties of the graphics frame. Refer to the description of **gi_startgframe()** for more information.

wprog is a Boolean value that, when set to **TRUE**, indicates that the CUSP button is to have CUSP program code added.

w*cap arguments are Boolean values that specify whether or not the CUSP button is to have captions. If a value of **TRUE** is specified for a **w*cap** argument. the respective **w*cap** return value will be non-**NULL**. **DocIC** functions may then be used to add text to each caption. Note the caption must eventually be freed by a call to **di_relcap()**.

If the **w*cap** arguments are set to **TRUE**, and a call to this function returns a valid button program handle, the returned handle must later be freed by a call to **gi_relbtnprog()**. **GraphicsIC** provides several functions that the user can call to add data to the CUSP program; refer to **gi_ap*btnprog()**, for information regarding these functions.

ret is a pointer to **ret_startnbtn**. It is a structure for the return information. It contains the following members:

```
gi_handle bfh;
gi_buttonprog prog;
di_caption tcap;
di_caption bcap;
di_caption lcap;
di_caption rcap;
```

The **bfh** argument is the handle to the newly created button. **prog** is a handle to the CUSP program code to be used by **gi_ap*btnprog** functions.

tcap, **bcap**, **lcap**, and **rcap** are as described above.

Once the desired graphics objects have been added to the frame, **gi_finishnbtn()** must be called to release the handle and resources allocated by the system.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

gi_startnbtn() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_ReadonlyDoc	Document opened in ReadOnly mode.
Doc_OutOfDiskSpace	Not enough disk space for the operation.
Doc_OutOfVM	Not enough virtual memory for the operation.
Doc_BadParm	One of the arguments specified is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

gi_startgframe(), gi_relbtnprog(), gi_ap*tobtnprog(), di_relcap(), gi_finishbtn(), gi_adffield(), gi_setgframeprops().

4. Table IC Library

ti__intro

NAME

ti__intro - introductory explanation of table functions

DESCRIPTION

TableIC functions are used to read the contents of a table, create a new table, or add information to an existing table.

A table is defined by three types of properties: table properties, column properties, and row properties. Table properties include the name of the table, a description of the table headers, and the number of columns and rows in the table; column properties include the division of columns and the alignment of text within columns; and row properties include information about how the text is to be aligned within a given row. The actual data of a table is included with the row information.

Table Building

The first step in generating a new table is usually a call to **ti__starttable()**. A call to this function will cause a table handle to be returned. The handle is a static variable that contains, in addition to table-related data, a pointer to the contents of the table. (Refer to **ti__*props** for a diagram that depicts the structure of a table.) It is this handle that is passed to other **ti__*()** functions as the means of identifying the table within the document and its associated properties. At this point, the row properties have default values and the contents of the table is nil. The contents and properties of each row are later added to the table via calls to other **ti__*()** functions.

To add data to a table, pass the table handle returned by **ti__starttable()** as an argument to **ti__appendrow()**. **ti__appendrow()** will add a row to the end of the table and insert the specified contents into that row. **ti__appendrow()** is to be called repeatedly until all the rows and their contents have been appended to the table. When the table is complete, call **ti__finishtable()** to finalize the structure of the table. Once finalized, **ti__finishtable()** returns an instance of the table, **di__ins**. This instance is comprised of only the rows and contents of the table frame; the remaining table properties, such as captions and border style, are added later via calls to related **DocIC** or **GraphicsIC** functions, such as **di__apaframe()** or **gi__adtable()**.

The preceding paragraphs describe how to create a new table and then add data to it. To add information to an existing table, call **ti__startextable()** rather than **ti__starttable()**. **ti__startextable()** is used to initialize a table in a document for editing. **ti__startextable()** is called with an instance of the table, **di__ins**, as an argument and returns a handle to that table. The table instance passed to **ti__startextable()** may be obtained by a call to **di__enumerate()**. When **di__enumerate()** is used to obtain the instance, the original document handle supplied as an argument to **di__enumerate()** must have originally come from **di__startap()**.

The table handle returned by **ti__startextable()** may then be passed as an argument to **ti__appendrow()** and **ti__finishtable()**.

Table Reading

The contents of a table are typically read by calling the **ti_enumtable()** function. This function requires an instance of a table, **di_ins**, and a set of three call-back procedures as arguments. The three call-back procedures are **ti_tableproc()**, **ti_columnproc()**, and **ti_rowproc()**.

The **ti_enumtable()** function calls **ti_tableproc()** and **ti_columnproc()** once while processing a table; these procedures extract the table and column properties. Since the contents of a table are stored with the rows, **ti_enumtable()** calls **ti_rowproc()** once for each row encountered in the table.

Properties

Table Properties

ti_tableprops describes the properties of a table and its headers. **ti_tableprops** contains the following members:

```
XString name;
unsigned nrows;
dp_bool fxrows;
unsigned ncols;
dp_bool fxcols;
dp_bool fillinbyrow;
dp_bool reptop;
dp_bool repbottom;
dp_bool deferon;
dp_bool vsblhd;
dp_bool rephd;
ti_hdalignment halign;
ti_valignment valign;
unsigned thdmgn;
unsigned bhdmgn;
ti_line bdrline;
ti_line dvrline;
ti_sortkeys *sortkeys;
```

name is an XString that specifies the name of the table.

nrows is an integer that specifies the number of rows in the table. This value is valid only upon reading by **ti_enumtable()**. **fxrows** is a Boolean value that specifies whether or not the user may change the number of rows in the table.

ncols is an integer that specifies the number of columns in the table. This value is valid only upon reading by **ti_enumtable()**. **fxcols** is a Boolean value that specifies whether or not the user may change the number of columns in the table.

fillinbyrow is a Boolean value that determines what happens when the user presses the NEXT key. If **fillinbyrow** is **TRUE**, pressing the NEXT key advances through the table one column at a time, and the table is expanded by rows. In this case, the number of columns is fixed and the number of rows can be either fixed or varying. If **fillinbyrow** is **FALSE**, then pressing the NEXT key advances through the table one row at a time, and the table is expanded by columns. In this case, the number of rows is fixed and the number of columns can be either fixed or varying.

vsblhd is a Boolean value that indicates whether or not there should be a visible header at the top of the table; **rephd, reptop, repbottom** are Boolean values that indicate whether or not to repeat the header, table top, or table bottom on every page when the table spans multiple pages.

deferon is a Boolean value that indicates whether the pagination operation will defer the table frame to the next page if it cannot fit on the current page. If **deferon** is **FALSE**, that portion of the table that will fit on the current page will be placed on that page, and the remainder will placed on successive pages.

halign and **valignm** are values of the type **ti_hdalignment** and **ti_valignment**, respectively. They specify the alignment of text within a header.

ti_hdalignment may have one of the following values:

```
HD_LEFT                 /* left */
HD_CENTER               /* center */
HD_RIGHT                /* right */
```

ti_valignment may have one of the following values:

```
VA_FTOP                 /* flush stop */
VA_CENTER               /* centered */
VA_FBOTTOM              /* flush bottom */
```

thdmgn and **bhdmgn** are integers that specify the amount of white space that should appear between above and below each header element. Values are specified in units of micas.

bdrline describes the table border (not the frame border), and **dvrline** describes the line between the header row and the rest of the table. A line may have a width anywhere from one pixel to six pixels. Both **bdrline** and **dvrline** are of the type **ti_line**.

ti_line is a structure whose members describe the properties of the line. It contains the following members:

```
ti_linestyle style;
ti_linewidth width;
```

ti_linestyle is an enumerated type that may have one of the following values:

```
LS_NONE                 /* none */
LS_SOLID                /* solid */
LS_DASHED               /* dashed */
LS_DOTTED               /* dotted */
LS_DOUBLE               /* double */
LS_BROKEN               /* broken */
```

ti_linewidth may have one of the following values:

```
LW_W1                   /* width of 1 pixel */
LW_W2                   /* width of 2 pixel  */
LW_W3                   /* width of 3 pixel  */
LW_W4                   /* width of 4 pixel  */
LW_W5                   /* width of 5 pixel  */
LW_W6                   /* width of 6 pixel  */
```

The argument **sortkeys** is a pointer of the type **ti_sortkeys**. It determines whether columns are sorted in ascending or descending order. It contains the following members:

```
unsigned length;
ti_sortkey *keys;
```

ti_sortkeys is comprised of an integer, **length**, and an optional array of **ti_sortkey**, where there may be one **ti_sortkey** specified for each table or column. A column must be divided-repeating in order to have sort keys. The integer specifies the number of columns comprising the table. **ti_sortkey** contains the following members:

```
XString name;
dp_bool ascend;
```

ti_sortkey consists of a string that specifies the column's name and a Boolean value that indicates whether to sort in ascending or descending order. A value of **TRUE** will result in an ascending sort order.

Column Properties

ti_colinfoseq describes the properties of all the columns in a table. **ti_colinfoseq** is a structure comprised of the following members:

```
unsigned length;
ti_colinforec *seq;
```

length is an integer that specifies the number of columns in the table.

seq is a pointer of the type **ti_colinforec**. Its members describe the properties of each column. The most complicated field in **ti_colinforec** is **hdentry**; all of the other fields correspond directly to the fields on the property sheet. The main column header properties are described below. The remaining column properties are described in the section titled *Other Column Properties*.

ti_colinforec contains the following members:

```
ti_hdentry hdentry;
XString name;
XString desc;
dp_bool divid;
unsigned subcols;
dp_bool repeat;
ti_colinfo subcolinfo;
ti_halignment alignment;
unsigned taboffset;
unsigned width;
unsigned lmgn;
unsigned rmgn;
dp_fldchoice type;
dp_bool req;
dp_lang lang;
XString format;
dp_bool stpskp;
XString range;
unsigned length;
XString skptext;
dp_skpchoice skpchoice;
XString fillin;
dp_fontruns *fillinruns;
ti_line line;
ti_sortkeys *sortkeys;
```

Column Header Properties

ti_hdentry is a structure whose members describe the textual content of each column header. The text in each column header may contain an unlimited number of font and paragraph properties. **ti_hdentry** contains the following members:

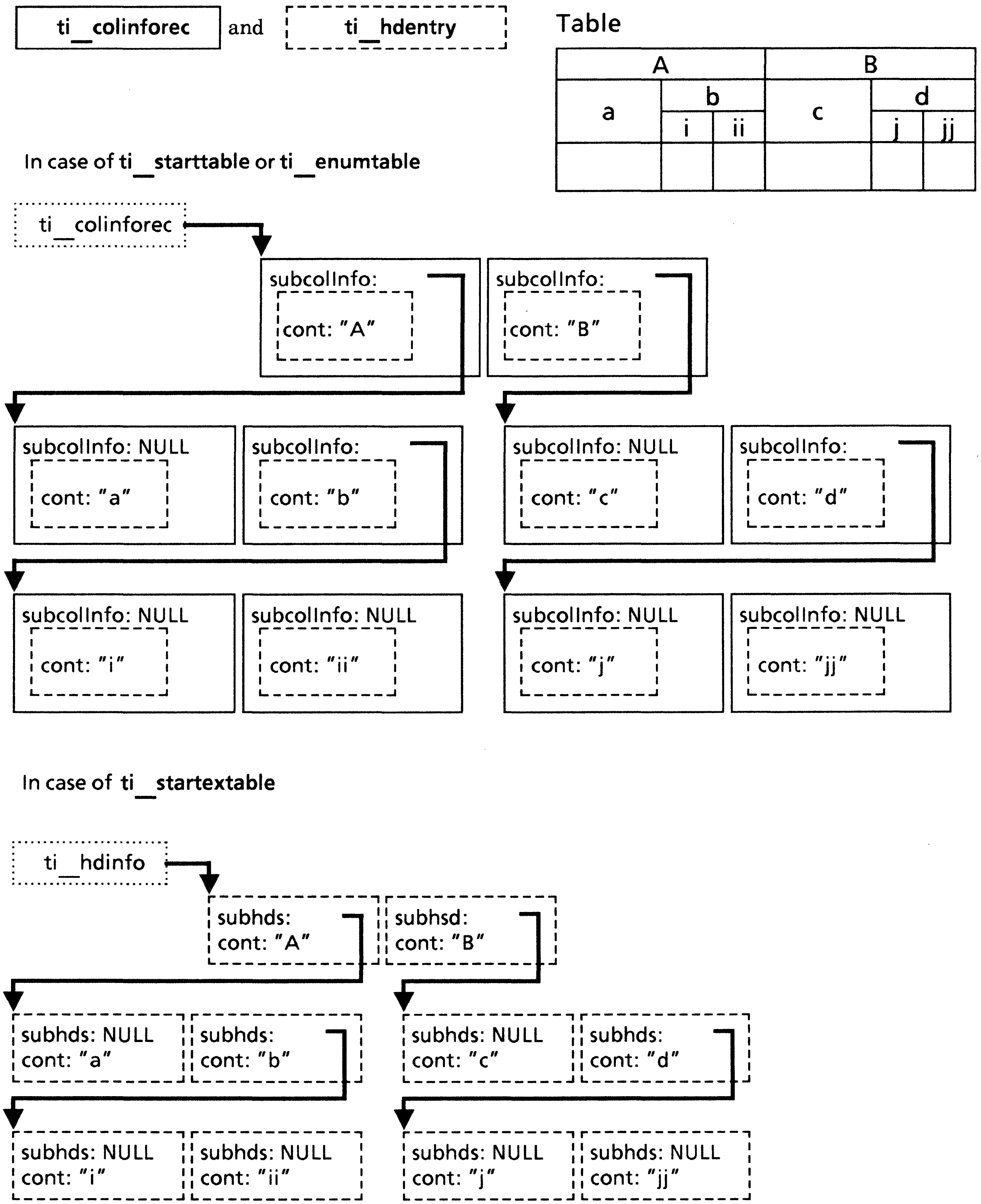
```
ti_hdinfo subhds;
ti_line line;
dp_bool hint;
ti_entry cont;
```

subhds is a value of the type **ti_hdinfo**. It is used to describe the headers of each subcolumn. This argument is applicable only if the column in question has been divided. **subhds** points to **ti_hdinfoseq**, a structure whose members define the number of columns in the table, including the **ti_hdentry** variable of each subcolumn. Each subcolumn may in turn be subdivided, in which case that subcolumn's **ti_hdentry.subhds** field will point to another array.

ti_hdinfoseq contains the following members:

```
unsigned length;
ti_hdentry *seq;
```

Refer to the figure below for a graphic description of the flow of **ti_*()** functions and the composition of headers and columns.



ti_colinforec and ti_hdentry

line is a value of the type **ti_line**. Its members describe the properties of the line that divides the header from subheaders. **line** contains the following members:

```
ti_linestyle style;
ti_linewidth width;
```

ti_linestyle is an enumerated type that may have one of the following values:

```
LS_NONE          /* none */
LS_SOLID         /* solid */
LS_DASHED        /* dashed */
LS_DOTTED        /* dotted */
LS_DOUBLE        /* double */
LS_BROKEN        /* broken */
```

ti_linewidth is an enumerated type that may have one of the following values:

```
LW_W1            /* width of 1 pixel */
LW_W2            /* width of 2 pixel  */
LW_W3            /* width of 3 pixel  */
LW_W4            /* width of 4 pixel  */
LW_W5            /* width of 5 pixel  */
LW_W6            /* width of 6 pixel  */
```

line is visible only when the column is subdivided.

hint is a Boolean value that indicates that the header is to contain only one line of text. Setting this value to **TRUE** will result in faster processing because it simplifies the calculation of header size. If **hint** is set to **TRUE** and then one or more lines of text are appended to the header text, the resulting header entry will display only the first line of text, though more text is present. If this situation occurs, it may be corrected by editing the text in the header, which will cause the Document Editor to recompute the header's height.

cont is a value of the type **ti_entry**. It is a union whose members describe the textual contents of a header and the access permissions to those contents. It contains the following members:

```
enum {
    READ = 0,
    WRITE = 1
    } mode;
union {
    di_text text;           /* NULL */
    wcont wcont;
    } u;
```

wcont contains the following members:

```
ti_fillintxtproc *proc;     /* NULL */
void *cdat;                 /* NULL */
```

When enumerating a table, all the header and row entry permissions will be set to **READ**. **di_text** may be called to enumerate the text. Upon completion, there is no need to call **di_reltext()**.

When creating a table, set all header and row entries to **WRITE**. **ti_fillintxtproc** call-back procedures may be invoked to fill the table with text. The value of **cdat** will be passed to **ti_fillintxtproc**.

Other Column Properties

name and **desc** are the name and description of the column as it would appear in the property sheet.

divid specifies whether the columns may be divided. **subcols** is the number of subcolumns; **repeat** indicates that subcolumns may have subrows, and **subcolinfo** is a recursive description of each subcolumn. **subcols, repeat,** and **subcolinfo** are ignored if **divid** is **FALSE**.

alignment describes the alignment of text within a column.

taboffset specifies where decimal tabs are to be set, relative to the margin. **taboffset** only applies if **alignment** is **THA_DECIMAL**. **taboffset** is specified in units of micas. Note that this is different than **dp_taboffset**, which is measured in units of points.

width is the width of a column; **lmgn** and **rmgn** are the margins of a column. If the column is divided, these parameters are ignored. These values are also specified in units of micas.

type indicates the type of contents to be placed in the column.

req indicates that data must be entered into a field before the user is permitted to proceed to another field in the table.

lang determines the format of the date and amount fields. It is used when items are added to the paragraph.

format allows the user to define a format to which the data in a column must conform.

stpskp controls the manner in which the skipping action of the SKIP button works. When set to **TRUE,** the skipping action will stop at the next entry in a column.

range defines a specific range of acceptable entries for the column. Once defined, an entry that does not match the criteria specified in **range** is not accepted.

length specifies the maximum number of characters to be entered into the column.

skptext and **skpchoice** define the conditions under which an entry may be skipped when the user presses NEXT.

fillin describes the fill-in rule for the column.

fillinruns describes the font properties of **fillin**.

line describes the properties of the vertical line to the right of the column.

sortkeys describes the sort keys for the column.

Row Content

ti_rowcont is a pointer to **ti_rowcontseq**. **ti_rowcontseq** is a pointer to a structure that describes the properties and contents of a row. It contains the following members:

```
unsigned tmgn;
unsigned bmgn;
ti_line line;
ti_valignment valign;
unsigned length;
ti_rowent *rowdat;
```

tmgn and **bmgn** are the row margins. That is, the margin above the top row and below the bottom row. **line** is the properties of the line separating the rows. **valign** specifies the alignment of text within a row. **rowdat** describes the contents of the row.

ti_rowent describes the textual content of a given row entry and contains the following members:

```
    ti_subrows *subrows;
    dp_bool hint;
    ti_entry cont;
```

ti_subrows describes the properties and contents of a subrow. If **subrows** is non-**NULL**, then the rest of the **ti_rowent** record is unused, since the textual information will be in each individual subrow record. **ti_subrows** contains the following members:

```
        unsigned length;
        ti_rowcont rows;
```

Note that subrows may exist only if the parent column is divided.

The remaining fields are as described in the header file.

ti_appendrow

NAME

ti_appendrow - append row

SYNOPSIS

```
#include "TableIC.h"

int
ti_appendrow(h, rc)
    ti_handle h;
    ti_rowcont rc;
```

DESCRIPTION

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

h is the value of **ti_handle**, an opaque variable that contains the table handle returned by an earlier call to either **ti_starttable()** or **ti_startextable()**.

rc is the value of the type **ti_rowcont**. It is a structure whose members specify the margins and alignment of the row, as well as its contents. It contains the following members:

```
unsigned tmgn;
unsigned bmgn;
ti_line line;
ti_valignment valign;
unsigned length;
ti_rowent *rowdat;
```

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_appendrow() will fail if the following is true:

Doc_TableTooTall	The specified table is too high to fit in the table frame.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_deffont, ti_defpara

NAME

ti_deffont, ti_defpara - default font and paragraph properties

SYNOPSIS

```
#include "DocICProps.h"
#include "TableIC.h"

int
ti_deffont(ret)
    dp_fontprops *ret;          /* Returned */

int
ti_defpara(ret)
    dp_paraprops *ret;          /* Returned */
```

DESCRIPTION

The **ti_deffont()** and **ti_defpara()** functions are used to assign default font and paragraph property values to the elements of a table.

These functions return **dp_fontprops** and **dp_paraprops**, respectively. The property structures that are returned contain default font or paragraph property values. The returned structures may be trapped and passed as arguments to the various table manipulation functions that require font or paragraph properties.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_def*() will fail if the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_enumtable

NAME

ti_enumtable - read table

SYNOPSIS

```
#include "DocIC.h"
#include "TableIC.h"

int
ti_enumtable(table, procs, cdat)
    di_ins table;
    ti_enumprocs *procs;
    void *cdat;                /* NULL */
```

CALLBACK PROCEDURE

```
ti_stop
ti_columnproc(cdat, columns)
    void *cdat;
    ti_colinfo columns;

ti_stop
ti_rowproc(cdat, cont)
    void *cdat;
    ti_rowcont cont;

ti_stop
ti_tableproc(cdat, props)
    void *cdat;
    ti_tableprops *props;
```

DESCRIPTION

The **ti_enumtable()** function is used to parse the contents of a table.

The table argument is the value of **di_ins**, an opaque variable that contains an instance of a table handle. The **procs** argument is the value of **ti_enumprocs**, a structure comprised of call-back procedures. Its members extract the properties of the table itself, and the properties of the columns and rows comprising the table. **ti_enumprocs** contains the following members:

```
ti_tableproc *table;           /* NULL */
ti_columnproc *column;         /* NULL */
ti_rowproc *row;               /* NULL */
```

table, **column**, and **row** are pointers to the respective procedures. **table** and the **column** are called once, but, since the data comprising a table is stored with the rows, **ti_enumtable()** calls **row** once for each row in the table.

Each call-back procedure returns a Boolean value. If the return value of **ti_stop** is TRUE, the enumeration will stop.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_enumtable() will fail if the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_finishtable

NAME

ti_finishtable - finish table

SYNOPSIS

```
#include "TableIC.h"

int
ti_finishtable(h, ret)
    ti_handle h;
    ret_ft *ret;                        /* Returned */
```

DESCRIPTION

The **ti_finishtable()** function is used to close the table currently being edited. This function must be called when no more edits are to be performed on the table.

h is the value of **ti_handle**, an opaque variable that contains the table handle returned by an earlier call to either **ti_starttable()** or **ti_startextable()**.

Once called, this function returns **ret_ft**, a structure that may be passed as the **cont** argument to **di_apaframe()** or as the **table** argument to **gi_adtable()**. **ret_ft** contains the following members:

```
di_ins table;
unsigned width;
unsigned height;
```

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_finishtable() will fail if the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_get*def

NAME

ti_get*def - get default properties

SYNOPSIS

```
#include "TableIC.h"

int
ti_getlinedef(line)
    ti_line *line;              /* Returned */

int
ti_getsortkeydef(sort)
    ti_sortkey *sort;           /* Returned */

int
ti_getcolinforecdef(col)
    ti_colinforec *col;         /* Returned */

int
ti_gethdentrydef(hdentry)
    ti_hdentry *hdentry;        /* Returned */

int
ti_getrowentdef(rowentry)
    ti_rowent *rowentry;        /* Returned */

int
ti_gettablepropsdef(props)
    ti_tableprops *props;       /* Returned */
```

DESCRIPTION

The **ti_getlinedef()** function is used to get default line properties . Their values are:

```
ti_linestyle style;          /* LS_SOLID */
ti_linewidth width;          /* LW_W1 */
```

The **ti_getsortkeydef()** function is used to get default sort key properties. Their values are:

```
XString name;                /* NULL */
dp_bool ascend;              /* TRUE */
```

The **ti_getcolinforecdef()** function is used to get default column properties . Their values are:

```
ti_hdentry hdentry;          /* null ti_hdentry */
XString name;                /* NULL */
XString desc;                /* NULL */
dp_bool divid;               /* FALSE */
unsigned subcols;            /* 0 */
dp_bool repeat;              /* FALSE */
ti_colinfo subcolinfo;       /* NULL */
ti_halignment alignment;     /* VA_CENTER*/
```

```
unsigned taboffset;          /* 0 */
unsigned width;              /* 2540 */
unsigned lmgn;               /* 0 */
unsigned rmgn;               /* 0 */
dp_fldchoice type;           /* FLD_ANY */
dp_bool req;                 /* FALSE */
dp_lang lang;                /* USE (USEnglish) */
XString format;              /* NULL */
dp_bool stpskp;              /* FALSE */
XString range;               /* NULL */
unsigned length;             /* 0 */
XString skptext;             /* NULL */
dp_skpchoice skpchoice;      /* SKP_EMPTY */
XString fillin;              /* NULL */
dp_fontruns *fillinruns;     /* NULL */
ti_line line;                /* LS_SOLID, LW_W2 */
ti_sortkeys *sortkeys;       /* NULL */
```

The ti_gethdentrydef() function is used to get default header entry properties . Their values are:

```
ti_hdinfo subhds;            /* NULL */
ti_line line;                /* LS_SOLID, LW_W2 */
dp_bool hint;                /* FALSE */
ti_entry cont;               /*  */
```

The ti_getrowentdef() function is used to get default row entry properties . Their values are:

```
ti_subrows *subrows;         /* NULL */
dp_bool hint;                /* FALSE */
ti_entry cont;               /*  */
```

The ti_gettablepropsdef() function is used to get default table properties . Their values are:

```
XString name;                /* NULL */
unsigned nrows;              /* 0 */
dp_bool fxrows;              /* FALSE */
unsigned ncols;              /* 0 */
dp_bool fxcols;              /* TRUE */
dp_bool fillinbyrow;         /* TRUE */
dp_bool reptop;              /* TRUE */
dp_bool repbottom;           /* TRUE */
dp_bool deferon;             /* FALSE */
dp_bool vsblhd;              /* TRUE */
dp_bool rephd;               /* TRUE */
ti_hdalignment halign;       /* HD_CENTER */
ti_valignment valign;        /* VA_CENTER */
unsigned thdmgn;             /* 0 */
unsigned bhdmgn;             /* 0 */
ti_line bdrline;             /* LS_NONE, LW_W1 */
ti_line dvrline;             /* LS_SOLID, LW_W4 */
ti_sortkeys *sortkeys;       /* NULL */
```

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ti_gettableprops

NAME

ti_gettableprops - get table props from name

SYNOPSIS

```
#include "DocIC.h"
#include "TableIC.h"
#include "XString.h"

int
ti_gettableprops(doc, name, ret)
    di_doc doc;
    XString name;
    ti_tableprops *ret;           /* Returned */
```

DESCRIPTION

The **ti_gettableprops()** function is used to extract the properties of a named table.

The **doc** argument is the value of **di_doc**, an opaque variable that contains the handle of the document which, in turn, contains the table in question.

The **name** argument is a text string that specifies the name of the table from which to extract the table properties.

This function returns a pointer to **ti_tableprops**, a structure that contains the properties of the named table. All the fields in the structure will accurately reflect the properties of the table except for the **name** field. It will be NULL. See **ti_starttable()** for a listing of **ti_tableprops** members.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_gettableprops() will fail if the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_maxelm

NAME

ti_maxelm - maximum table elements

SYNOPSIS

```
#include "TableIC.h"

int
ti_maxelm(ret)
    unsigned *ret;              /* Returned */
```

DESCRIPTION

The **ti_maxelm()** function is used to estimate the number of table cells that could reside in a document that does not contain other structures. The value that is returned may be used to estimate how big a table may be created within the document.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_maxelm() will fail if the following is true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_startextable

NAME

ti_startextable - open an existing table

SYNOPSIS

```
#include "DocIC.h"
#include "DocICProps.h"
#include "TableIC.h"

int
ti_startextable(table, hi, rowsource, deleterow, ret)
    di_ins table;
    ti_hdinfo hi;                 /* NULL */
    unsigned rowsource;           /* 0 */
    dp_bool deleterow;            /* TRUE */
    ti_handle *ret;               /* Returned */
```

DESCRIPTION

The **ti_startextable()** function is used to add data to an existing table. Data is added by appending rows which contain the data to the existing table.

The **table** argument is the value of **di_ins**, an opaque variable that points to an instance of a table.

hi is the value of **ti_hdinfo**, a pointer to the structure **ti_hdinfoseq**. This structure contains the following members:

```
unsigned length;          /* Number of ti_hdentry */
ti_hdentry *seq;          /* Array of ti_hdentry */
```

ti_hdentry is a structure that is specified as an array, with one **ti_hdentry** specification per table column header. The members of **ti_hdentry** specify the contents and appearance of a column header in the table. **ti_hdentry** contains the following members:

```
ti_hdinfo subhds;
ti_line line;
dp_bool hint;
ti_entry cont;
```

length is an integer that specifies the number of **ti_hdentry** entries in the array of **ti_hdentry**.

If **hi** is **NULL**, then the existing column headers are used.

rowsource is the index of a row in the table. The properties of the specified row will be extracted and applied as the default properties to each new row. The range of the index is between 0 and *n*, inclusive, where *n* is the number of rows. All properties of the new row, except for the horizontal alignment, are taken from the row specified in **rowsource**. The horizontal alignment of each element of the new row is the same as that of the first row.

deleterow is a Boolean value that indicates whether the table contents should be deleted before adding new information. When set to **TRUE**, all the rows and their contents are deleted from the table, except for header information.

Like **ti_starttable()**, **ti_startextable()** returns **ti_handle**, an opaque variable that contains a table handle. It may then be passed as an argument to **ti_appendrow()** and **ti_finishtable()**.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_startextable() will fail if the following is true:

Doc_ReadonlyDoc	The document is read-only.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

ti_starttable

NAME

ti_starttable - create a new table

SYNOPSIS

```
#include "DocIC.h"
#include "TableIC.h"

int
ti_starttable(doc, props, col, ret)
    di_doc doc;
    ti_tableprops *props;
    ti_colinfo col;
    ti_handle *ret;          /* Returned */
```

DESCRIPTION

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

The **doc** argument is the value of **di_doc**, an opaque variable that contains the document handle for the document to which the table will be added.

The **props** argument is a pointer to **ti_tableprops**, a structure whose members specify the properties of the new table. These properties include the name of the table, the number of columns and rows to be assigned, the alignment of the table within the frame, and so on. **ti_tableprops** contains the following members:

```
XString name;
unsigned nrows;
dp_bool fxrows;
unsigned ncols;
dp_bool fxcols;
dp_bool fillinbyrow;
dp_bool reptop;
dp_bool repbottom;
dp_bool deferon;
dp_bool vsblhd;
dp_bool rephd;
ti_hdalignment halign;
ti_valignment valign;
unsigned thdmgn;
unsigned bhdmgn;
ti_line bdrline;
ti_line dvrline;
ti_sortkeys *sortkeys;
```

col is the value of **ti_colinfo**, a pointer to a structure of the type **ti_colinfoseq**. **ti_colinfoseq** is an array of **ti_colinforec**, with one **ti_colinforec** per each column in a table. It specifies the properties of a column, such as headers, width, margins, and the text to put in each columns. **ti_colinforec** contains the following members:

```
ti_hdentry hdentry;
XString name;
XString desc;
```

```
dp_bool divid;
unsigned subcols;
dp_bool repeat;
ti_colinfo subcolinfo;
ti_halignment alignment;
unsigned taboffset;
unsigned width;
unsigned lmgn;
unsigned rmgn;
dp_fldchoice type;
dp_bool req;
dp_lang lang;
XString format;
dp_bool stpskp;
XString range;
unsigned length;
XString skptext;
dp_skpchoice skpchoice;
XString fillin;
dp_fontruns fillinruns;
ti_line line;
ti_sortkeys *sortkeys;
```

This function returns **ret**, a pointer to **ti_handle**. **ti_handle** is an opaque variable that contains a table handle.

ti_starttable() will raise a **Doc_DocumentFull** error if the table and header row can not fit in the document. This error is raised when there is no more room to add an object (e.g., a table) into the specified document. The size of a VP document may be as large as disk space allows but the structured portions may not exceed 255 disk pages. One disk page is comprised of 512 bytes.

RETURN VALUE

If the call is successful 0 is returned, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

ti_starttable() will fail if one or more of the following are true:

Doc_DocumentFull	No more room in the document.
Doc_TableTooWide	The specified table is too wide to fit in the document.
Doc_TableHeaderTooTall	The specified headers are too tall.
Doc_BadParm	One of the specified arguments is invalid.
Doc_IllegalHandle	The specified handle is illegal.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

5. Desktop Library

dsktp__intro

NAME

dsktp__intro - introductory description of Desktop functions

DESCRIPTION

Desktop functions are used to manipulate existing document files and folders located on the Desktop or to add new files and folders to the Desktop. **dsktp_*()** functions are used to copy or delete existing files, make folders, and more. The most important aspect of **Desktop** functions is that they allow the interaction between files on the desktop and the editing functions of other interfaces, such as those in **DocIC** and **GraphicsIC**.

di_start() is called as the first step in the document generation process. Afterwards, the contents of a document may be appended using **DocIC** and **GraphicsIC** functions. Lastly, **di_finish()** is called to finalize the document. **di_finish()** returns a reference, or handle, to the newly created document. This reference may be passed in calls to other **dsktp_*()** functions. Typically, this reference is passed as an argument in a call to **dsktp_movedoc()**. The result of this function is to take the new file, which currently exists only in a buffer, and place it on the Desktop. Once on the Desktop, the new file may be manipulated like any other document.

When manipulating an existing document, **dsktp_enumerate()** or **dsktp_getdocref()** is called as the first step in the document editing process. The reference that is returned may then be passed as an argument to **di_open()** or **di_startap()**. These functions return a handle, **di_doc**, that may be passed to document editing functions, such as those contained within in the Document IC Library and the Graphics IC Library. The last step in the editing process is to indicate that the document is finished by a call to, either, **di_close()** or **di_finish()**. The finished document still resides in a temporary buffer. To move it from the buffer onto the Desktop, **dsktp_movedoc()** must be called.

dsktp_checkuser

NAME

dsktp_checkuser - verify the VP user identity

SYNOPSIS

```
#include "Desktop.h"
#include "DocICProps.h"

int
dsktp_checkuser(user, passwd, ret)
    char *user;
    char *passwd;
    dp_bool *ret;           /* Returned */
```

DESCRIPTION

The **dsktp_checkuser()** function is used to verify the identity of a user accessing the Desktop. This function checks both the user name and password.

The **user** argument is a string that indicates the user to be validated. It is specified in the form: *name:domain:organization.*

The **passwd** argument is a string that specifies the VP password of the person identified in the **user** argument.

The **ret** argument is where the results of **dsktp_checkuser()** are placed. **ret** will have a Boolean value. The value will be **TRUE** only when both the name and password supplied match the current VP logon user name and password.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_checkuser() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

SEE ALSO

dsktp_getaccess()

dsktp_copydoc

NAME

dsktp_copydoc - duplicate a document

SYNOPSIS

```
#include "Desktop.h"

int
dsktp_copydoc(ref, new)
    dsktp_docref ref;
    dsktp_docref new;            /* Returned */
```

DESCRIPTION

The **dsktp_copydoc()** function is used to copy a document and return a handle to the duplicate document.

The **ref** argument is an opaque variable of the type **dsktp_docref**. It is a reference to the document that is to be copied. **ref** was returned by an earlier call to **dsktp_getdocref()** or **dsktp_enumerate()**.

The **new** argument is also an opaque variable of the type **dsktp_docref**. This argument defines the structure of the return information into which will be placed the handle information for the duplicate document.

Note that this function does not generate an icon for the duplicate document, only a handle to it. If an icon is desired for the duplicate document, use the **dsktp_movedoc()** function. At which time, a unique name may be assigned to the duplicate document.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_copydoc() will fail if one or more of the following are true:

Doc_BadParm One of the specified arguments is invalid.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

DT_FileChanged The file has been modified during program execution such that it cannot be used.

DT_FileDamaged The file has been internally damaged.

DT_FileInUse The specified file is in use by another application.

DT_FileNotFound The file was not found in the expected context.

DT_Illegal One of the arguments to the function call is invalid.

DT_FileNotUnique The directory already contains a file of the same name (if the UniquelyNamedContents of Folder Properties is set to **TRUE**) or the same name and version (if the UniquelyNamedContents of Folder Properties is set to **FALSE**).

DT_LoopInHierarchy The directory is the same as, or a descendant of, the file being moved or copied.

DT_MediumFull	There is not enough space on the appropriate file service to satisfy the request.
DT_NoAccessRight	Reading and/or writing to the Desktop is not allowed.

SEE ALSO

dsktp_movedoc()

dsktp__deletedoc

NAME

dsktp__deletedoc - delete a document

SYNOPSIS

```
#include "Desktop.h"

int
dsktp_deletedoc(ref)
   dsktp_docref ref;
```

DESCRIPTION

The **dsktp_deletedoc()** function is used to remove a VP document from off the desktop, from within a folder on the desktop, or from a nested folder.

The **ref** argument is an opaque variable of the type **dsktp_docref**. It is a handle, or pointer, to the document to be moved. **ref** was returned by an earlier call to **dsktp_getdocref()**.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_deletedoc() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
DT_FileChanged	The file has been modified during program execution such that it cannot be used.
DT_FileDamaged	The file has been internally damaged.
DT_FileInUse	The specified file is in use by another application.
DT_FileNotFound	The file was not found in the expected context.
DT_Illegal	One of the arguments to the function call is invalid.
DT_MediumFull	There is not enough space on the appropriate file service to satisfy the request.
DT_NoAccessRight	Reading and/or writing to the Desktop is not allowed.

dsktp_enumerate

NAME

dsktp_enumerate - enumerate documents

SYNOPSIS

```
#include "Desktop.h"
#include "XString.h"

int
dsktp_enumerate(pattrn, path, depth, list)
    XString pattrn;
    XString path;
    unsigned short depth;
    dsktp_reflist *list;          /* Returned */
```

DESCRIPTION

The **dsktp_enumerate()** function is used to list the documents in a folder, a nested folder, or on the desktop that match a specified criteria.

The **pattrn** argument is a text string that specifies the pattern to be used in searching for files. Two wildcard characters are supported: * (asterisk) and # (pound). The asterisk character matches zero or more characters in the file name; the pound character matches any single character in the file name. To use the asterisk and pound characters literally, so that they have no special significance, they must be preceded by a single quote.

The **path** argument is a text string that specifies the full path name of the folder or nested folder in which to begin the search. To specify the desktop, set the value of **path** to **NULL**. A version number may be appended to the path name. If a version number is omitted from the path name, the most recent version is assumed.

The **depth** argument is an integer that indicates the levels of the folder hierarchy in which to descend during the search for documents. The search begins with the folder specified in the **path** argument. A value of 1 indicates that only the folder specified in **path** is to be searched.

The **list** argument is a pointer to the returned list and is of the type **dsktp_reflist**. It points to a structure whose members specify the number of objects in the list and a pointer to the handle containing the list itself. **dsktp_reflist** contains the following members:

```
int len;
dsktp_docref *refs;
```

len is an integer that indicates the total number of documents that matched the search criteria. **ref** is a pointer to **dsktp_docref**, an array containing a reference to each document that matched the search criteria.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_enumerate() will fail if one or more of the following are true:

Doc_BadParm One of the specified arguments is invalid.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

DT_FileChanged The file has been modified during program execution such that it cannot be used.

DT_FileDamaged The file has been internally damaged.

DT_FileInUse The specified file is in use by another application.

DT_FileNotFound The file was not found in the expected context.

DT_Illegal One of the arguments to the function call is invalid.

DT_MediumFull There is not enough space on the appropriate file service to satisfy the request.

DT_NoAccessRight Reading and/or writing to the Desktop is not allowed.

dsktp_getaccess

NAME

dsktp_getaccess - obtain the desktop information

SYNOPSIS

```
#include "Desktop.h"

int
dsktp_getaccess(ac)
    dsktp_access *ac;          /* Returned */
```

DESCRIPTION

The **dsktp_getaccess()** function is called to ascertain the status and access permissions of the Desktop.

The **ac** argument is a pointer of the type **dsktp_access**. **dsktp_access** is an enumerated variable that is set by the call and may have one of the following values:

DT_NONE	/* Both ReadAccess and WriteAccess are **FALSE** */
DT_READ	/* ReadAccess is **TRUE**, WriteAccess is **FALSE** */
DT_WRITE	/* ReadAccess is **FALSE**, WriteAccess is **TRUE** */
DT_READWRITE	/* Both ReadAccess and WriteAccess are **TRUE** */
DT_LOGOFF	/* The Desktop is not opened */

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_getdocref() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.

dsktp_getdocprops

NAME

dsktp_getdocprops - obtain properties of a file

SYNOPSIS

```
#include "Desktop.h"

int
dsktp_getdocprops(ref, props)
    dsktp_docref ref;
    dsktp_docprops *props;          /* Returned */
```

DESCRIPTION

The **dsktp_getdocref()** function is used to obtain the properties of a document on the Desktop. The properties associated with a Desktop document are name, version, size, creation date, creator and type.

The **ref** argument is a variable of the type **dsktp_docref**. It is a reference to the file whose properties are to be returned.

The **props** argument is a pointer to **dsktp_docprops**. It is called to set the properties associated with a Desktop document. **dsktp_docprops** contains the following members:

```
XString name;              /* file name */
unsigned short vers;       /* version */
unsigned short size;       /* size in disk pages */
dsktp_date date;           /* creation date */
XString username;          /* created by */
dsktp_doctype type;        /* file type */
```

dsktp_date contains the following members:

```
unsigned short year;       /* year expressed in four digits */
unsigned short month;      /* [1-12] */
unsigned short day;        /* [1-31] */
unsigned short hour;       /* [0-23] */
unsigned short minute;     /* [0-59] */
unsigned short second;     /* [0-59] */
```

dsktp_doctype may have one of the following values:

```
DT_DOC                     /* document */
DT_FOLDER                  /* folder */
DT_OTHER                   /* other than document and folder */
```

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_getdocref() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.

Doc_TimeOut Inter-process communication has exceeded the maximum allowed time.

DT_FileChanged The file has been modified during program execution such that it cannot be used.

DT_FileDamaged The file has been internally damaged.

DT_FileInUse The specified file is in use by another application.

DT_FileNotFound The file was not found in the expected context.

DT_Illegal One of the arguments to the function call is invalid.

DT_MediumFull There is not enough space on the appropriate file service to satisfy the request.

DT_NoAccessRight Reading and/or writing to the Desktop is not allowed.

dsktp__getdocref

NAME

dsktp__getdocref - obtain a document handle

SYNOPSIS

```
#include "Desktop.h"
#include "XString.h"

int
dsktp_getdocref(name, vers, srcpath, ref)
    XString name;
    unsigned short vers;
    XString srcpath;
    dsktp_docref ref;          /* Returned */
```

DESCRIPTION

The **dsktp_getdocref()** function is used to acquire a handle, or reference, to a document on the desktop. The returned handle may then be passed as an argument to other related functions .

A document is referenced as name-version pair. The **name** argument is a text string that specifies the name of the document to which a handle is desired. The **ver** argument is an unsigned short integer that indicates the version number of the document. If set to to **NULL**, the most recent version is assumed.

The **srcpath** argument is a text string that specifies the desktop, folder, or nested folder in which the desired document resides. The format for specifying a folder or nested folder is the same as currently used to designate paths in NSFiling: folder1!v1/folder2!v2../folder*N*!v*N*. Separator characters, such as "/" and "!", should be escaped when they occur within folder names. They are escaped by preceding them by a single quote. Wildcards are not supported. If a version number is omitted from the path string, the most recent version is searched. To access a document that is on the desktop, set the value of **srcpath** to **NULL**.

The **ref** argument is the return value and is of the type **dsktp_docref**. It is an array of four unsigned integers whose elements identify the document.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_getdocref() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
DT_FileChanged	The file has been modified during program execution such that it cannot be used.
DT_FileDamaged	The file has been internally damaged.
DT_FileInUse	The specified file is in use by another application.
DT_FileNotFound	The file was not found in the expected context.
DT_Illegal	One of the arguments to the function call is invalid.

DT_MediumFull There is not enough space on the appropriate file service to satisfy the request.

DT_NoAccessRight Reading and/or writing to the Desktop is not allowed.

SEE ALSO

di_open(), di_finish()

dsktp__makefolder, dsktp__deletefolder

NAME

dsktp__makefolder, dsktp__deletefolder - create a new folder or remove an existing folder

SYNOPSIS

```
#include "Desktop.h"
#include "XString.h"

int
dsktp_makefolder(name, dstpath, vers)
    XString name;
    XString dstpath;
    unsigned short *vers;          /* Returned */

int
dsktp_deletefolder(name, vers, srcpath)
    XString name;
    unsigned short vers;
    XString srcpath;
```

DESCRIPTION

The **dsktp_makefolder()** function is used to create a folder on the desktop or within an existing folder.

The **name** argument is a text string that specifies the name of the folder to be created. The **dstpath** argument is the full path to an existing folder, or nested folder, in which the new folder is to be placed. To specify the desktop, set the value of **dstpath** to **NULL**.

This function returns **vers**, an unsigned short integer that indicates the version number of the new folder.

The **dsktp_delete()** folder function is used to remove a folder from within another folder or from the desktop. The **name** argument is a text string that specifies the folder to be deleted. The **vers** argument is an integer that specifies the version number of the folder to be deleted.

The **srcpath** argument is a text string that specifies the desktop, folder, or nested folder in which the document to be deleted resides. The format for specifying a folder or nested folder is described in **dsktp_getdocref()**. To specify the desktop, set the value of **srcpath** to **NULL**.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_*folder() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
DT_FileChanged	The file has been modified during program execution such that it cannot be used.
DT_FileDamaged	The file has been internally damaged.
DT_FileInUse	The specified file is in use by another application.

DT_FileNotFound The file was not found in the expected context.

DT_Illegal One of the arguments to the function call is invalid.

DT_FileNotUnique The directory already contains a file of the same name (if the UniquelyNamedContents of Folder Properties is set to **TRUE**) or the same name and version (if the UniquelyNamedContents of Folder Properties is set to **FALSE**).

DT_MediumFull There is not enough space on the appropriate file service to satisfy the request.

DT_NoAccessRight Reading and/or writing to the Desktop is not allowed.

SEE ALSO

dsktp_getdocref()

dsktp_movedoc

NAME

dsktp_movedoc - move or rename a document

SYNOPSIS

```
#include "Desktop.h"
#include "XString.h"

int
dsktp_movedoc(ref, dstpath, name, vers)
    dsktp_docref ref;
    XString dstpath;
    XString name;
    unsigned short *vers;        /* Returned */
```

DESCRIPTION

The **dsktp_movedoc()** function is used to move a document to a folder, a nested folder, or the desktop. This function may also be used as a means to rename a document.

The **ref** argument is the value of **dsktp_docref**, an opaque variable that is a reference, or pointer, to the document to be moved. **ref** was returned by an earlier call to **dsktp_getdocref()** or **dsktp_copydoc()**.

The **dstpath** argument is a text string that specifies the full path name of the folder or nested folder in which to place the document. Refer to **dsktp_getdocref()** for a description of how to specify a full path. To specify the desktop, set the value of **dstpath** to **NULL**.

The **name** argument is a text string that specifies the name of the moved document. If left **NULL**, the same name is assigned to the moved document as the source document. To rename a document, specify a different **name** but the same **dstpath** as that of the source document. If version numbers are omitted from the path string, the most recent versions are used.

The **vers** argument is a pointer to an integer that indicates the version number ultimately assigned to the moved document.

RETURN VALUE

0 is returned if the call is successful, otherwise -1 is returned. The function **getsigno()** is used to get the reason for the failure.

ERRORS

dsktp_movedoc() will fail if one or more of the following are true:

Doc_BadParm	One of the specified arguments is invalid.
Doc_TimeOut	Inter-process communication has exceeded the maximum allowed time.
DT_FileChanged	The file has been modified during program execution such that it cannot be used.
DT_FileDamaged	The file has been internally damaged.
DT_FileInUse	The specified file is in use by another application.
DT_FileNotFound	The file was not found in the expected context.
DT_Illegal	One of the arguments to the function call is invalid.

DT_FileNotUnique The directory already contains a file of the same name (if the UniquelyNamedContents of Folder Properties is set to **TRUE**) or the same name and version (if the UniquelyNamedContents of Folder Properties is set to **FALSE**).

DT_LoopInHierarchy The directory is the same as, or a descendant of, the file being moved or copied.

DT_MediumFull There is not enough space on the appropriate file service to satisfy the request.

DT_NoAccessRight Reading and/or writing to the Desktop is not allowed.

SEE ALSO

dsktp_getdocref()

6. XString Library

XString__intro

NAME

XString__intro - introductory description of XString functions

DESCRIPTION

Characters and strings in the VP Document Editor are structured differently than their UNIX counterparts. XString library functions manipulate characters and strings for use by other Document Interfaces Toolkit functions and for conversion between VP and UNIX structures. The XString library is also the means by which multinational characters and strings may be manipulated. The XString library supports a set of multinational character codes that adhere to the XCCS(Xerox Character Code Standard).

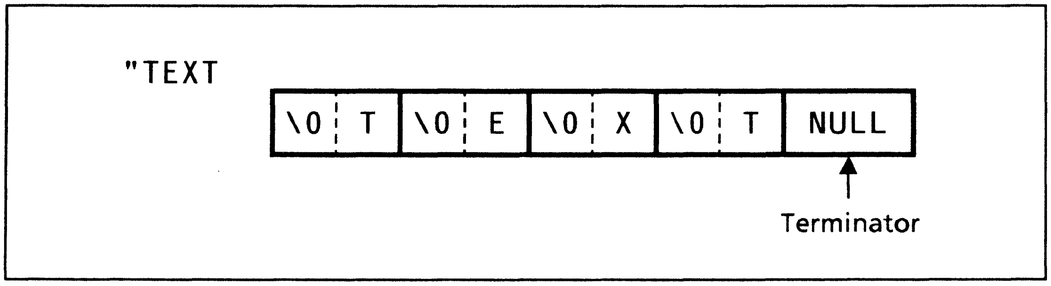
The primary XString library functions perform basic services such as string copying, appending, separating, comparing, and searching. These functions are very similar to conventional UNIX C string handling functions. For example, **XStrcat()** is analogous to **strcat()**. The difference being, **XStrcat()** is used to concatenate strings that are in the XCCS format and **strcat()** concatenates ASCII strings.

The XString library provides functions for conversion between XCCS 8-bit encoded strings and 16-bit encoded XStrings, as well as conversion between ASCII strings and XStrings. Currently, the XString library does not support conversion to and from EUC (Extended UNIX Code or JIS (Japanese Industrial Standard).

XString data structure

An XChar is an unsigned short integer (16 bits). Internally, it is comprised of two 8-bit bytes, where the first byte defines the XCCS character set and the second byte defines the XCCS character code. The character code determines the character's appearance. The character set determines the character's semantic meaning.

An XString is a simple array of XChar with a a 16-bit **NULL** code (0x0000) at the end of the XString to act as the terminator.



Structure of XString

All string creation and editing functions in XString library, except **XStrncpy()**, assume that the resulting string is terminated by a **NULL** character. Furthermore, XCC8 (Xerox Character Code for 8 bit characters) is defined for a data structure of 8-bit encodings in the XCCS. XCC8 is analogous to ByteSequence in Mesa XString.

XCharset, XCharcode, XCharmake

NAME

XCharset, XCharcode, XCharmake - determine the character set or code used, or define character/code pairs

SYNOPSIS

```
#include "XString.h"

unsigned char
XCharset(xc);
    XChar xc;

unsigned char
XCharcode(xc);
    XChar xc;

XChar
XCharmake(set, code)
    char set;
    char code;
```

DESCRIPTION

The **XChar*** functions are used to invoke a character set having specific visual properties. That is, these functions invoke a set of characters, as well as the style in which these characters are represented. These character styles are referred to as character codes.

A character code is defined as any numeric code that represents a graphic character, a rendering character, or a control character. The definition of a graphic character and a control character are self-evident. A rendering character is defined as one of the following:

- a non-conventional representation of a control code
- a sequence of graphic characters (i.e., ligature or accented character)
- a context-dependent alternate representation of a graphic character (e.g., initial, medial, or final form for an alphabet such as Arabic)
- a "variant" representation of a graphic character

In effect, a character code is the static representation of textual content. A sequence of numeric character codes is referred to as a string. Textual information is stored and transmitted as a sequence of numeric character codes.

A character set is 256 blocks, with each block containing 256 codes. Each 16 bit character code consists 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.

The **XCharset()** function is used to retrieve the XChar character set (the higher 8 bits) of a character. The **xc** argument is the character for which the character set information is to be retrieved.

The **XCharcode()** function is used to retrieve the XChar character code (the lower 8 bits). The **xc** argument is the character for which the character code information is to be retrieved.

XCharmake() is used to create an XChar character of an existing character, based upon definitions contained within the Xerox Character Code Standard (XCCS). The variables, **set** and **code**, specify the character set and the character code, respectively, that are to be used in creating the new characters. For example,

```
xchar = XCharmake(0,'x');
```

results in an xchar being created for x.

RETURN VALUE

XCharset() and **XCharcode()** return the character code of **xc**. **XCharmake()** returns XChar character.

XStrcat, XStrncat

NAME

XStrcat, Xstrncat - append to a string

SYNOPSIS

```
#include "XString.h"

XString
XStrcat(xs1, xs2)
    XString xs1, xs2;

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

DESCRIPTION

The **XStrcat()** function is used to concatenate one string to the end of another string. The **xs1** argument is the string to which the other string is to be appended. That is, **xs1** is the first portion of the concatenated string. It is the programmer's responsibility to ensure that sufficient storage is allocated for the data to fill **xs1**. The **xs2** argument is the string to be appended. It is the second, or tailing portion of the concatenated string.

The **XStrncat()** function is used to copy a specific number of characters from one string and append them to the end of another string. The **xs1** argument is the string to which the copied characters are to be appended. It is the programmer's responsibility to ensure that sufficient storage is allocated for the data to fill **xs1**. The **xs2** argument is the string from which a specific number of characters are to be copied and then appended to **xs1**. The **n** argument is the number of characters that are to be copied from **xs2** and appended to **xs1**.

RETURN VALUE

XStrcat() returns **xs1**. **XStrncat()** returns **xs1**.

SEE ALSO

XStrcpy(), XStrncpy(), XStrdup()

XStrcmp, XStrncmp, XStrcasecmp, XStrncasecmp

NAME

XStrcmp, XStrncmp, XStrcasecmp, XStrncasecmp - compare strings

SYNOPSIS

```
#include "XString.h"

int
XStrcmp(xs1, xs2)
    XString xs1, xs2;

int
XStrncmp(xs1, xs2, n)
    XString xs1, xs2;
    int n;

int
XStrcasecmp(xs1, xs2)
    XString s1, xs2;

int
XStrncasecmp(xs1, xs2, n)
    XString xs1, xs2;
    int n;
```

DESCRIPTION

The **XStr*cmp()** functions are used to compare two strings. They return negative, zero, or positive integers if the first string in the argument list is less than, equal to, or greater than the second string in the argument list. For example, in comparing the following strings:

```
xs1: abcdef
xs2: abcxyz
```

XStrcmp() compares the characters in **xs1** against **xs2** on a one-by-one basis. Upon reaching the fourth character in **xs2**, a difference would be discovered. 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.

Comparisons are done in the order specified in the Xerox Character Code Standard (XCCS).

The **XStrcmp()** function is used to compare two strings, **xs1** and **xs2**.

The **XStrncmp()** function is used to compare a portion of one string against another string. The **xs1** argument is the string from which the first **n** characters are to be compared against the first **n** characters of the string specified in the **xs2** argument.

The **XStrcasecmp()** function is used to compare two strings, while ignoring the case of ASCII characters. That is, upper-case characters are equal to the lower-case equivalent characters. The **xs1** and the **xs2** arguments are the two strings to be compared. Non-ASCII characters will be compared in accordance to the order specified by the XCCS.

The **XStrncasecmp()** function is used to compare a portion of one string against another string, while ignore the case of ASCII characters. The **xs1** argument is the string from which the first **n** characters are to be compared against the first **n** characters of the string specified in in the **xs2** argument. Non-ASCII characters will be compared in accordance to the order specified by the XCCS.

RETURN VALUE

positive integer:	**xs1 > xs2**
zero:	**xs1 = xs2**
negative integer:	**xs1 < xs2**

SEE ALSO

XStrlexcmp(), XStrnlexcmp(), XStrcaselexcmp(), XStrncaselexcmp()

XStrcpy, XStrncpy, XStrdup

NAME

XStrcpy,XStrncpy,XStrdup - copy string

SYNOPSIS

```
#include "XString.h"

XString
XStrcpy(xs1, xs2)
    XString xs1, xs2;

XString
XStrncpy(xs1, xs2, n)
    XString xs1, xs2;
    int n;

XString
XStrdup(xs1)
    XString xs1;
```

DESCRIPTION

The **XStrcpy()** function is used to copy a specific string into a virtual memory storage area, defined by the user. The argument **xs2** is the text string to be copied. The argument **xs1** is a pointer to the storage area that is to receive the string. It is the programmer's responsibility to ensure that sufficient storage is allocated for the data to fill **xs1**.

The **XStrncpy()** function is used to copy a specific number of characters in a text string. The argument **xs2** is the text string to be copied. The argument **xs1** is a pointer to the storage area that is to receive the string. It is the programmer's responsibility to ensure that sufficient storage is allocated for the data to fill **xs1**.

The argument **n** is an integer that specifies the number of character in the **xs2** argument to copy. Copying begins with the first letter of the text string and proceeds to the last. If the number of characters to copy, **n**, is greater than than the length of the string, **xs2**, then the entire string, including the **NULL** character, will be copied. If the number of characters to copy is less than or equal to the number of characters in the string, the string will be copied and the terminating **NULL** character truncated.

The **XStrdup()** function is used to copy a text string into a storage area and return a pointer to that area. The **xs1** argument is the string to be copied. Memory space for the copy is reserved by malloc. If malloc fails in memory reservation, a **NULL** pointer is returned.

RETURN VALUE

XStrcpy() and **XStrncpy()** return **xs1**. The return value of **XStrdup()** is **xs1**, a pointer to the duplicate string. A **NULL** pointer is returned if malloc fails in memory allocation.

SEE ALSO

XStrcat(), XStrncat()

XStrlen

NAME

XStrlen - string length

SYNOPSIS

```
#include "XString.h"

int
XStrlen(xs);
    XString xs;
```

DESCRIPTION

The **XStrlen()** function is used to determine the logical length of an XString character. The returned value will specify the number of characters in the string as an integer. The returned character length will not include the terminating **NULL** character.

The **xs** argument is the string whose length is to be determined.

RETURN VALUE

The character length of **xs**.

XStrlexcmp, XStrnlexcmp, XStrcaselexcmp

NAME

XStrlexcmp, XStrnlexcmp, XStrcaselexcmp - compare strings lexicographically

SYNOPSIS

```
#include "XString.h"

int
XStrcaselexcmp(xs1, xs2, sortorder)
    XString xs1, xs2;
    SortOrder sortorder;

int
XStrlexcmp(xs1, xs2, sortorder)
    XString xs1, xs2;
    SortOrder sortorder;

int
XStrncaselexcmp(xs1, xs2, sortorder, n)
    XString xs1, xs2;
    SortOrder sortorder;
    int n;

int
XStrnlexcmp(xs1, xs2, sortorder, n)
    XString xs1, xs2;
    SortOrder sortorder;
    int n
```

DESCRIPTION

The **XStr*lexcmp()** functions are used to lexicographically compare two strings. They return negative, zero, or positive integers if the first string in the argument list is lexicographically less than, equal to, or greater than the second string in the argument list. Comparisons are done in accordance to the order specified by **sortorder**.

The **XStrlexcmp()** function is used to compare two strings, **xs1** and **xs2**, according to the order specified in **sortorder**. **sortorder** is described in detail in the document, *Multinational Programming Considerations*.

The **XStrnlexcmp()** function is used to compare the first **n** characters of **xs1** against **xs2**, based upon the value of **sortorder**.

The **XStrcaselexcmp()** function is used to compare two strings, while ignoring the case of ASCII characters and while sorting the character strings in the order specified in **sortorder**. Upper-case characters will be equal to the lower-case equivalent characters. The **xs1** and the **xs2** arguments are the two strings to be compared. Non-ASCII characters will be compared in accordance to the specified lexicographical order defined by **sortorder**.

The **XStrncaselexcmp()** function is used to compare a portion of one string against another string, while ignoring the case of ASCII characters. The **xs1** argument is the string from which the first **n** characters are to be compared against the string specified in the **xs2** argument, according to the specified sort order, **sortorder**.

sortorder is the value of **SortOrder**, an enumerated type that may contain one of the following values:

STANDARD, DANISH, SPANISH, SWEDISH

Please refer to the table below for a description as to the category each language falls into:

Language	SortOrder
Canadian (English)	Standard
Canadian (French)	Standard
Danish	Danish
Dutch	Standard
Finnish	Swedish
French	Standard
German	Standard
Italian	Standard
Norwegian	Danish
Portuguese	Standard
Spanish	Spanish
Swedish	Swedish
United Kingdom	Standard
United States	Standard

RETURN VALUE

1: **xs1** > **xs2**
0: **xs1** = **xs2**
-2: RPC function, **clnt_create()** failed.
-3: RPC function, **clnt_call()** failed.
-4: UNIX standard function, **gethostname()** failed.
-5: The length of **xs1** or **xs2** exceeds 8192 bytes.

SEE ALSO

XStrcmp(), XStrncmp(), XStrcasecmp(), XStrncasecmp()

XStrchr, XStrrchr, XStrpbrk

NAME

XStrchr, XStrrchr, XStrpbrk- search for a character

SYNOPSIS

```
#include "XString.h"

XString
XStrchr(xs, xc)
    XString xs;
    XChar xc;

XString
XStrrchr(xs, xc)
    XString xs;
    XChar xc;

XString
XStrpbrk(xs1, xs2)
    XString xs1, xs2;
```

DESCRIPTION

The **XStrchr()** function is used to parse a string in search of a specific character. It starts the search at the beginning of the string and proceeds towards the end. The **xs** argument is the string to be searched. The **xc** argument is the character to be found in the string. If the specified character is found, a pointer to the first occurrence of the character is returned. If the specified character is not found, a **NULL** pointer is returned.

The **XStrrchr()** function, like **XStrchr()**, searches a string for a character. It starts the search at the end of the string and proceeds towards the beginning. If the specified character is found, a pointer to the first occurrence of the character in the string is returned. If the specified character is not found, a **NULL** pointer is returned.

For example, to find character ”x” in the following example:

abcxdefxg

XStrchr() will return a pointer to the ”x” which is the third character from the left. **XStrrchr()** will return a pointer to the ”x” which is the second character from the right.

The **XStrpbrk()** function is used to search a string for the occurrence of any character contained within another string. The **xs1** argument is the string to be searched. The **xs2** argument is the string from which each character is extracted and then compared against each character in **xs1**. The first character in **xs2** is parsed, placed in a buffer and then compared against each character in **xs1**. The comparison stops upon the first match. The system then returns a pointer to the first occurrence of the matching character in **xs1**. If the specified character is not found, a **NULL** pointer is returned.

RETURN VALUE

A pointer to the character's location, if it is found. A **NULL** pointer, if the character is not found.

SEE ALSO

XStrsch()

XStrsch

NAME

XStrsch - search for a string

SYNOPSIS

```
#include "XString.h"

XString
XStrsch(xs1, xs2)
    XString xs1, xs2;
```

DESCRIPTION

The **XStrsch()** function is used to determine if a string is contained within another string. It starts the search at the beginning of the string and proceeds towards the end. The **xs1** argument is the main string. The **xs2** argument is the string that you would like to find in **xs1**. If the search is successful, the system returns a pointer to the first occurrence of **xs2** in **xs1**. Otherwise, a **NULL** pointer is returned.

RETURN VALUE

A pointer to the string, if it is found. A **NULL** pointer, if it is not found.

SEE ALSO

XStrchr(), XStrrchr(), XStrpbrk()

XStrsep

NAME

XStrsep - separate a string into tokens

SYNOPSIS

```
#include "XString.h"

XString
XStrsep(xs1, xs2, xc)
    XString    xs1, xs2;
    XChar      *xc;
```

DESCRIPTION

The **XStrsep()** function is used to separate a string into tokens based upon one or more delimiter characters. The **xs1** argument is the string to be separated. The **xs2** argument is a string containing one or more delimiter characters. Each character within **xs2** is considered as a delimiter. Separator characters may be standard delimiters, such as ",", "!", ":", and ";", or they be any desired text characters. The **xc** argument is a pointer to the returned delimiter character.

This function returns a pointer to the first character of the first token and returns the delimiter character to **xc**. If the delimiter characters specified in **xs2** can not be found in string **xs1**, then the system returns the entire string as one token and sets **xc** to **NULL**. When **xs1** can not be further separated into tokens, a **NULL** pointer is returned.

After completing a call to **XStrsep()**, the original string specified as the argument **xs1** will no longer exist.

RETURN VALUE

Pointers to the first character of each separated token.

XStrfromASC, XStrtoASC

NAME

XStrfromASC, XStrtoASC - convert ASCII and XString strings

SYNOPSIS

```
#include "XString.h"

XString
XStrfromASC(xs, s)
    XString xs;
    char *s;

XString
XStrtoASC(xs, s, c)
    XString xs;
    char *s;
    char c;
```

DESCRIPTION

The **XStrfromASC()** function is used to convert the ASCII string pointed to by **s** to XString **xs**. The return value will be **xs**. It is the programmer's responsibility to insure that sufficient storage space is allocated for **xs**. **xs** will require (2× **s** byte length) + 2.

Basically, this function does not convert control coeds. If a control or 8-bit code (a code that belongs to the group on the shift-out side) is in the ASCII string, **s**, it will be simply extended to a 16-bit code and copied into **xs**. If the resultant code is identical to an XCCS code of the character set 0, it will be expressed as a VP character in VP documents. If it is identical to a control code like a tab in VP, it will be used as is. If it is not defined in VP, it will be expressed as a black square. An exception is 0xFF, which will be converted to 0x007F and copied into **xs**. It is the user's responsibility to process these codes correctly. Note the following codes.

0x09 (11B)	→	Tab (Tab)
0x0D (15B)	→	New Line (NewLine)
0x1D (35B)	→	New Paragraph (NewPara)
0x89 (211B)	→	Paragraph Tab (ParaTab)
0x87 (207B)	→	Page Number
0x8E (216B)	→	Table of contents mark (left)
0x8F (217B)	→	Table of contents mark (right)

The **XStrtoASC()** function is used to convert the XString **xs** into an ASCII string **s**. During the conversion, characters that do not have ASCII equivalents are replaced by the character signified by **c**. This function returns 0, if all the characters in **xs** were successfully converted to ASCII. Otherwise, it returns the number of non-ASCII characters in **xs**.

It is the programmer's responsibility to assure that sufficient storage is allocated for **s**. **s** will require (**xs** character length + 1 byte) for storage.

RETURN VALUE

xs is returned by **XStrfromASC**. **XStrtoASC** returns the number of non-ASCII characters in **xs**.

XStrfromXCC8, XStrtoXCC8

NAME

XStrfromXCC8, XStrtoXCC8 - convert between XCCS 8-bit encoded string and XString

SYNOPSIS

```
#include "XString.h"

XString
XStrfromXCC8(xs, xcc8, len, prefix)
    XString xs;
    XCC8 xcc8;
    int len;
    int prefix;

int
XStrtoXCC8(xs, xcc8)
    XString xs;
    XCC8 xcc8;
```

DESCRIPTION

The **XStrfromXCC8()** function is used to convert an XCCS 8-bit encoded string into an XString string. The **xs** argument is the storage area in which to place the converted XString string. The **xcc8** argument is the XCCS 8-bit encoded string that is to be converted. The **len** argument specifies the length in bytes of **xcc8**. The **prefix** argument should be set to 0 if **xcc8** is a standard 8-bit encoded string. The encoded string is considered to be "standard" if the first character begins with the character set 0 or with character set select (0xff). If the first character in **xcc8** begins with a character code that indicates the use of a non-zero character set, the value of the **prefix** argument should also use that same character set. **prefix** should be -1 if the first character of **xcc8** begins with a 16-bit code. A successful conversion returns **xs**. An unsuccessful conversion returns a **NULL** pointer.

To calculate sufficient storage resources for **xs**, allow (2 * **xcc8** byte length) + 2 bytes .

The **XStrtoXCC8()** function is used to convert an XString string into a compact XCCS 8-bit encoded string. The **xs** argument is the value of the XString string to be converted. The **xcc8** argument is the return value that is to contain the XCCS 8-bit encoded string.

In the XCCS system, a 16-bit encoded representation is possible by placing two character set selects (0xff) plus 0 (total of three bytes) at the point where the 16-bit encoding representation starts. Therefore, **XStrtoXCC8()** first compares the length of the 8-bit and 16-bit ([0xff,0xff,0x0] at the head of **xs**) encoding representations that **XStrtoXCC8()** would get after converting **xs**. After which, **XStrtoXCC8()** converts **xs** into **xcc8** in the shorter representation.

The first byte of the converted **xcc8** begins, either, with a character code having a character set 0, or with character set select 0 (0xff). The return code will be the byte length of the converted **xcc8** string.

To calculate sufficient storage resources for **xcc8**, allow (2 * **xs** character length) + 3 bytes for storage.

XCC8 requires data structures of 8-bit encodings in a XCCS format. **XCC8** is analogous to ByteSequence in Mesa XString.

RETURN VALUE

XStrtoXCC8() returns the byte length of **xcc8**. **XStrfromXCC8()** returns a **NULL** pointer, if **xcc8** encoding is invalid or **xs** is invalid, otherwise it returns **xs**.

7. Document IC Signals

getsigno

NAME

getsigno - retrieve the number of an error

SYNOPSIS

#include "Signals.h"

int
getsigno()

DESCRIPTION

When a function returns -1, **getsigno()** is called to determine the cause for the failure.

The **getsigno()** function takes no arguments and its return value indicates the reason for the failure.

The following is a list of error numbers and the corresponding text names, as specified in "**Signals.h**":

/* Signals from Document IC Toolkit operations */

4096 (0x1000)	Doc__ContainerFull Insufficient space for appending to this container.
4097 (0x1001)	Doc__DocumentFull Insufficient space in the document.
4098 (0x1002)	Doc__ReadonlyDoc Document opened in ReadOnly mode.
4099 (0x1003)	Doc__OutOfDiskSpace Insufficient disk space for the operation.
4100 (0x1004)	Doc__OutOfVM Insufficient virtual memory for the operation.
4101 (0x1005)	Doc__ObjIllegalInCont Attempted to add an object of an unsupported type to a container.
4102 (0x1006)	Doc__BadParm One of the arguments specified is invalid.
4103 (0x1007)	Doc__Unimpl This function is not supported.
4104 (0x1008)	Doc__OutOfRoomForGraphics Insufficient space in the document to insert graphics objects.
4105 (0x1009)	Doc__TableTooWide The specified table is too wide to fit in the document.

4106 (0x100a) Doc_TableTooTall
The specified table is too tall to fit in the document.

4107 (0x100b) Doc_TableHeaderTooTall
The specified headers are too tall.

4108 (0x100c) Doc_TimeOut
Timeout has occurred during inter-process communication.

4109 (0x100d) Doc_IllegalHandle
The handle specified is invalid.

4110(0x100e) Doc_NoAccessRight
Reading and/or writing to the document is not allowed.

/* Signals from XString operations */

8192 (0x2000) XS_Illegal
The specified XString is invalid.

/* Signals from Desktop operations */

16384 (0x4000) DT_FileChanged
While the function was executing, the file changed in such a way that execution could not continue. This condition may occur, for example, when **dsktp_enumerate()** is called and the order of the files contained in a folder or on the desktop changes.

16385 (0x4001) DT_FileDamaged
A file is internally damaged in some way.

16386 (0x4002) DT_FileInUse
The specified file is in use by another application.

16387 (0x4003) DT_FileNotFound
A file was not found in the expected context.

16388 (0x4004) DT_Illegal
One of the arguments passed to the desktop interface is invalid.

16389 (0x4005) DT_FileNotUnique
The directory already contains a file with the same name (if the UniquelyNamedContents of Folder Properties is set to **TRUE**) or the same name and version (if the UniquelyNamedContents of Folder Properties is set to **FALSE**).

16390 (0x4006) DT_LoopInHierarchy
The directory is the same as, or a descendant of, the file being moved or copied.

16391 (0x4007) DT_MediumFull
There is not enough space on the appropriate file service to satisfy the request.

16392(0x4008) DT_NoAccessRight
Reading and/or writing to the desktop is not allowed.

/* Signals from implementation failures */

32767 (0x7fff) IMPL_SIG
An unimplemented module has been encountered.

/* Place holder for unidentified signals */

32766 (0x7ffe)	OTHER__SIG The default signal that is displayed when an error occurs that cannot be addressed by any of the preceding signals.

RETURN VALUE

The return value of **getsigno()** is the reason for the most recent failure of all but XNS functional calls.

8. XNS Library

XNS__intro

NAME

XNS__intro - introductory description of XNS interchange functions

DESCRIPTION

The XNS toolkit library is the means by which UNIX/C programmers may interface with Xerox XNS servers. The XNS toolkit library contains functions, referred to as stubs, that allow calls to be made to the clearinghouse server, the authentication server, file servers, the print server, mail servers, and the Gateway Access Protocol (GAP) server. Xerox System Integration Standards contain protocol information on these servers. The standards described in these books, however, define the protocols with respect to the Xerox Mesa language. C programmers may still benefit from the descriptions in these books because Mesa and C functions are similar in many respects. Function calls in C require the same parameters as the equivalent Mesa functions. These parameters are of the same type in both programming languages. The exception is, the C representation of a function contains two extra parameters. They are **__Connection** and **__BDTprocptr**.

__Connection

Every XNS function called by a C application must contain a value for the parameter **__Connection**. This parameter is the courier connection number of the XNS server to which the C application is attempting communication. Therefore, for example, depending upon the XNS server number entered, it is possible to direct a C application to communicate with a specific printer.

The number to be supplied as the value of the **__Connection** parameter may be obtained by entering the following code in the C application:

```
COURIERFD  connected;
char       *hostnameptr;

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

In the example code above, **connected** is the return value of the **cour__establish__conn** function supplied by libcourier.a (a library that you must link with to use XNS functions).

hostnameptr is a string that contains the name of the desired server. For example, if you have access to the Xerox organization and the Sunnyvale domain, and you wish to access a printer called BCobain in that domain, then:

hostnameptr = "BCobain:Sunnyvale:Xerox"

would be be the proper format for specifying the printer. If a connection to the host does not occur, an error message will be printed.

MUST BE AN IP MASTER

Some XNS functions do not require a valid value for **_Connection**. Those functions requiring a valid **_Connection** value are described as appropriate. The remaining functions should be set to **NULL**.

_BDTprocptr

Every XNS function also requires a value for the parameter **_BDTprocptr**. For those functions that transfer bulk data, this parameter is the name of the function that performs the bulk data transfer. This bulk data function is created by the C programmer. As an example, to print a UNIX file the C code may contain the following call:

NO (JOHN JOHNSON)

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

Two parameters of special importance are **SendSource** and **BulkData1_immediateSource**. **SendSource** is the name of the user-defined C function that sends print data from the UNIX environment to the XNS printer defined by the **printconnected** parameter, the courier connection number for the printer. The code you write to define **SendSource** may be as follows:

```
int

    SendSource (bdtconnection)
    COURIERFD   bdtconnection;
{
    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;
}
```

When transferring bulk data, another parameter of type **Sink** or **Source** must also be supplied. These two types are bulk data types. They direct the function to source data from the UNIX environment or sink data to the UNIX environment. To send data from UNIX to XNS, use **BulkData1_immediateSource**. To retrieve data from XNS back to UNIX, use **BulkData1_immediateSink**.

A valid value for this parameter is only required if the function transfers bulk data. If it does not, set the value of this parameter to **NULL**.

Error Handling

The code to trap errors generated by XNS functions must be defined by the user. Each XNS function has a specified set of errors that it may return. The Standards book for the respective protocol explains each error. This manual lists the possible errors each function may return. Two errors not described in either the Gray Book or this manual are courier-generated errors: REJECT_ERROR and SYSTEM_ERROR. These two errors may be generated by any XNS function. The following code may be inserted in the C program to catch these errors:

```
#include    <courier/except.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: Some random 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;
```

When an error occurs, the XNS function will return a code number and sometimes a problem number. The above code switches on the error code number in order to print out the user-defined error message. If the error also returns a problem number, you can determine the cause of the error by calling **CourierErrArgs()**. Refer to the respective Standards book for more details.

Be sure to include *except.h* in the application. This header file defines the macros DURING, HANDLER, and END__HANDLER.

Header Files

Each XNS service has two particular header files associated with it: *[service]__de.h* and *[service].h*, where *[service]* represents the name of the service. For example the printing service, which would be *Printing3__de.h* and *Printing3.h*. Your application should include one or the other, but not both. The *[service]__de.h* header files simplifies typing. It has define statements that eliminate the need for prefixing function and error statements with the service name. *[service].h* is the "raw" header file. If you include this header file, you must prefix the name of the service to each function or error name in the application. For example, the function **ChangeStrongKey()** may be specified in one of two ways: If the header file used is *Authentication2.h*, the function must be specified as **Authentication2_ChangeStrongKey()**. If the header file used is *Authentication2.de.h*, then the function may be specified as **ChangeStrongKey()**.

Authentication2_ChangeStrongKey, _ChangeSimpleKey

NAME

ChangeStrongKey, ChangeSimpleKey – change a user's strong or simple key

SYNOPSIS

```
#include <courier/Authenti2_de.h>
#include <courier/except.h>

void
ChangeStrongKey(_Connection, _BDTprocptr, credentials, verifier, newKey)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    Block newKey;

void
ChangeSimpleKey(_Connection, _BDTprocptr, credentials, verifier, newKey)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    Cardinal newKey;
```

DESCRIPTION

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

The strong **credentials** and **verifier** arguments identify the client for whom the key is to be changed. The **newKey** argument is the strong key that has been encrypted using the ECB mode of DES, and the conversation key that is contained in the credentials. The encryption and decryption of the strong key is performed by user-defined code.

The **ChangeSimpleKey()** function is used to change a simple key that is registered with the Authentication Service.

The simple **credentials** and **verifier** arguments identify the user for whom the simple key is to be changed. The **newKey** argument is the unencrypted new key that is to be registered. The **newkey** must be hashed by the user.

Use of these functions is contingent upon how the Internet is administered. If you are unable to change a strong or simple key via remote function calls, it may be due to the Internet administrative rules.

RETURN VALUE

These functions return **void**.

ERRORS

Reports [AuthenticationError[problem], CallError[problem]]

SEE ALSO

CreateStrongKey(), CreateSimpleKey()

Authentication2__CheckSimpleCredentials

NAME

CheckSimpleCredentials – verify a user's identity

SYNOPSIS

```
#include <courier/Authenti2_de.h>
#include <courier/except.h>

void
CheckSimpleCredentials(__Connection, __BDTprocptr, credentials, verifier)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **CheckSimpleCredentials()** function is used to verify that the correct password has been submitted to the Authentication Service. The Authentication Service compares the simple key that is registered for the initiator against the simple key in the **verifier**. The **credentials** are used to specify the Clearinghouse in which the initiator is registered.

The **credentials** and **verifier** arguments must be the simple **credentials** and **verifier** of the initiator. Simple credentials are the initiator's **ThreePartName**, specified as a text string. Simple verifier is the result of a hashing algorithm applied by the Authentication Service upon the initiator's password.

RETURN VALUE

This function returns a structure called **CheckSimpleCredentialsResults**. Its one member is a Boolean value. A value of **TRUE** indicates that the simple key registered for the initiator and the simple key specified in the verifier match.

ERRORS

Reports [AuthenticationError[problem], CallError[problem]]

Authentication2__CreateStrongKey, __CreateSimpleKey

NAME

CreateStrongKey, CreateSimpleKey – register a new strong or simple key

SYNOPSIS

```
#include <courier/Authenti2__de.h>
#include <courier/except.h>

void
CreateStrongKey(__Connection, __BDTprocptr, credentials, verifier, name, key)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    ThreePartName name;
    Key key;

void
CreateSimpleKey(__Connection, __BDTprocptr, credentials, verifier, name, key)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    ThreePartName name;
    Cardinal key;
```

DESCRIPTION

The **CreateStrongKey()** function is used to register a strong key with the Authentication Service.

The **credentials** and **verifier** specified must be the strong credentials and strong verifier of a privileged user. The Authentication protocol for these two is described below. **name** is the user name as known by the Clearinghouse. The key is the strong key to be registered with the Authentication Service. It will be encrypted in the ECB mode of DES , using the conversation key contained in the credentials.

Strong credentials consist of data which has been encrypted using the National Bureau of Standards' Data Encryption Standard (DES). A **key** is an array comprised of 4 16-bit wide words, where the least significant bit is assigned as the parity bit, thus leaving 56 bits for unconstrained data. The least significant bit of each octet is set so as to make the parity of each octet odd.

The **CreateSimpleKey()** function is used to register a new simple key with the Authentication Service. A simple key is a simple password that has been hashed according to the algorithm specified in the Xerox Authentication Protocol manual. Only a privileged user may register a new key.

The **credentials** and **verifier** arguments must be the strong **credentials** and verifier of a privileged user. The **name** argument specifies the intended user of the **key**. The **key** argument is the unencrypted key that is to be registered.

RETURN VALUE

These functions return **void**.

ERRORS

Reports [AuthenticationError[problem], CallError[problem]]

SEE ALSO

ChangeStrongKey(), ChangeSimpleKey(), GetStrongCredentials()

Authentication2__DeleteStrongKey, __DeleteSimpleKey

NAME

DeleteStrongKey, DeleteSimpleKey – delete a user's strong or simple key

SYNOPSIS

```
#include <courier/Authenti2_de.h>
#include <courier/except.h>

void
DeleteStrongKey(_Connection, _BDTprocptr, credentials, verifier, name)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    ThreePartName name;


void
DeleteSimpleKey(_Connection, _BDTprocptr, credentials, verifier, name)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Credentials credentials;
    Verifier verifier;
    ThreePartName name;
```

DESCRIPTION

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

The **credentials** and **verifier** arguments must be the strong **credentials** and **verifier** of the key's owner or of a privileged user. The **name** argument specifies the user for whom the key is to be deleted.

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

The **credentials** and **verifier** arguments must be the simple credentials and verifier of the possessor or of a privileged user. The **name** argument specifies the user for whom the key is to be deleted.

RETURN VALUE

These functions return **void**.

ERRORS

Reports [AuthenticationError[problem], CallError[problem]]

SEE ALSO

CreateStrongKey(), CreateSimpleKey()

Authentication2_GetStrongCredentials

NAME

GetStrongCredentials – acquire privileged user permission

SYNOPSIS

```
#include <courier/Authenti2_de.h>
#include <courier/except.h>

GetStrongCredentialsResults
GetStrongCredentials(_Connection, _BDTprocptr, initiator, recipient, nonce)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName initiator;
    ThreePartName recipient;
    LongCardinal nonce;
```

DESCRIPTION

The **GetStrongCredentials()** function is used to create credentials in order to prove one's identity to a specified communications partner (i.e., recipient). Once created, the privileged user can act on behalf of any user within the same organization and domain. To get StrongCredentials you must have a strong key registered with the Authentication Service and you must know how to decrypt the results.

It is sometimes necessary to authenticate oneself to the Authentication Service, such as when modifying a strong key or fetching credentials through a proxy. To authenticate oneself, you must supply the name of the Authentication Service. Since any instance of the Authentication Service may be specified, the service is accessible through a "wellknown" name. This wellknown name may be used regardless of the instance actually being accessed. The wellknown name of the Authentication Service is Authentication Service:CHServers:CHServers.

A sender, called the **initiator**, attempts to authenticate itself to a receiver, called the **recipient**. To do this the sender contacts the Authentication Service, via this function, and supplies to the Service the names of both parties and a random number, called a **nonce**. The **nonce** is a check mechanism that insures the validity of the Authentication Service. If the sender is properly registered with the Authentication Service, this function will return credentials, the **nonce**, the receiver's name and conversation key. All four are encrypted. The decrypted credentials are used by other functions, such as **DeleteSimpleKey()**. The conversation key is not passed to any function. It is used to encrypt verifiers that are later passed to those functions requiring strong verifiers.

The **initiator** argument is the distinguishing name, or alias, of the user that wishes to be authenticated. The **recipient** argument is the distinguishing name, or alias, of the recipient to whom the initiator is proving his identity.

RETURN VALUE

This function returns a structure called **GetStrongCredentialsResults**. Its one member, **credentialsPackage**, is of type **T_r14_2_2**. It has been encrypted with the initiator's key. Once decrypted with the initiator's key, it will contain credentials that have been encrypted with the recipient's key, of which only the recipient may decrypt. It will also contain a **nonce**, the recipient's name, and a conversation key. Once the **credentialsPackage** has been decrypted it is possible for the initiator to view the nonce, the recipient's name, and conversation key.The initiator may not view the credentials because it is still encrypted with the recipient's strong key.

ERRORS

Reports [CallError[problem]]

Clearinghouse2_AddGroupProperty

NAME

AddGroupProperty - add a group type property to an object

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

AddGroupPropertyResults
AddGroupProperty(_Connection, _BDTprocptr,name, newProperty, membership, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName name;
    LongCardinal newProperty;
    BulkData1_Descriptor membership;
    Authenticator agent;
```

DESCRIPTION

The **AddGroupProperty()** function is used to add a new group type property to an object. The value of a group type property is understood by the Clearinghouse to be a sequence of Clearinghouse names called members.

A Clearinghouse object is comprised of three parts: a property number (ID), a property type, and a value. A property is primarily used to hold a network location, or a list of other object names. Given an object name and a property number, the Clearinghouse will return the value of that property, which will be either a block of data (if the property type is *item*), or a list of names (if the property type is *group*). The Clearinghouse does not inspect *item* properties, therefore they may consist of any data the client wishes. The *group* property, on the other hand, is inspected and recognized by the Clearinghouse, therefore each group property must contain a sequence of Clearinghouse names called *members*.The *name* argument is the Clearinghouse name of the object. It may be in the form of either the actual name of the object or its alias. UNIX wildcards may not be used in specifying any part of the object name.

The **name** argument is comprised of three strings that identify the organization, domain, and name of an object. Wildcards may not be used to specify any portion of this argument. It may be a distinguished name or an alias.

The **newProperty** argument identifies the group type property that is to be added to an object.

The **membership** argument is a Bulk Data Transfer parameter that specifies the source that supply the list of names in accordance to the Bulk Data Transfer Protocol. This list of names provides the initial value of the new group type property. That is, the group type property is initialized with zero or more members as specified by the source. The data sent via **membership** is of type **SegmentOfThreePartName**. Wildcard characters may occur in any part of each name, but the characters will not have wildcard significance. They will be interpreted as regular characters.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **AddGroupPropertyResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object that received the new group type property.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

ListProperties()

Clearinghouse2__AddItemProperty

NAME

AddItemProperty -- add a item type property to an object

SYNOPSIS

```
#include <courier/Clearing2__de.h>
#include <courier/except.h>

AddItemPropertyResults
AddItemProperty(__Connection, __BDTprocptr, name, newProperty, value, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    LongCardinal newProperty;
    Item value;
    Authenticator agent;
```

DESCRIPTION

The **AddItemProperty** function is used to add a property of a specified value to an object. The property value will be of type Item. Item type properties are not inspected by the Clearinghouse and therefore do not have to adhere to Clearinghouse rules. An object may have up to 250 properties associated with it.

A Clearinghouse object is comprised of three parts: a property number (ID), a property type, and a value. A property is primarily used to hold a network location, or a list of other object names. Given an object name and a property number, the Clearinghouse will return the value of that property, which will be either a block of data (if the property type is item), or a list of names (if the property type is group). The Clearinghouse does not inspect item properties, therefore they may consist of any data the client wishes. The group property, on the other hand, is inspected and recognized by the Clearinghouse, therefore each group property must contain a sequence of Clearinghouse names called members.The name argument is the Clearinghouse name of the object. The name may be either the actual name of an object or an alias. UNIX wildcards may not be used in specifying any part of the object name.

If an attempt is made to add a property that already exists, even if it has a different value, the attempt will be ignored. Use **ChangeItem()** to change the value of an existing item property.

The **name** argument is the name of the object to which the property will be added. It may be either the actual name of the object or an alias. UNIX wildcards may not be used to specify any portion of the **name** argument. The **newProperty** argument is an integer that identifies the property to be added. The **value** property is the initial value, or data, to be assigned the new property.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **AddItemPropertyResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object to which the item type property was added.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

AddGroupProperty(), ListProperties()

Clearinghouse2__AddMember, __AddSelf

NAME

AddMember - add a member to a group type property
AddSelf - add the user to a group type property

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

AddMemberResults
AddMember(__Connection, __BDTprocptr, name, property, newMember, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    ThreePartName newMember;
    Authenticator agent;


AddSelfResults
AddSelf(__Connection, __BDTprocptr, name, property, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    Authenticator agent;
```

DESCRIPTION

The **AddMember()** function is used to add a new member to a group type property of an object. The **AddSelf()** function is used to add the user identified by the **agent** argument to a group property of an object.

The value of a group property is understood by the Clearinghouse to be a sequence of Clearinghouse names called members. The new member may be a distinguished name, an alias, or the name of a Clearinghouse object that does not currently exist. The name of the member does not have to be registered with the Clearinghouse at the time of calling this function, though the object must be registered.

The **name** argument specifies the object to which the new member is to be added. It is of type **ThreePartName**. Its members, **organization**, **domain**, and **object**, identify the object in question. UNIX wildcards may not be used to specify any part of the name. If the object name encountered is an alias, it is dereferenced before it is processed.

The **property** argument specifies the property number of the property to which the new member will be added. The **newMember** argument identifies the new member. It is specified as being of type **ThreePartName**.

The **agent** argument is a structure of type **Authenticator**. Its two members contain the client's credentials and verifier. In the case of **AddSelf**, **agent** identifies the user and verifies the user's credentials. In the case of **AddMember**, it simply verifies the user's credentials. The new user is identified by the **newMember** argument.

RETURN VALUE

AddMember() and **AddSelf()** return structures called **AddMemberResults** and **AddSelfResults**, respectively. They both have one member, **distinguishedObject**, which is of type **ThreePartName**. It is the distinguished name of the object to whose group type property the member was added.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

DeleteMember(), AddGroupProperty()

Clearinghouse2_ChangeItem

NAME

ChangeItem - modify the value of an item type property

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

ChangeItemResults
ChangeItem(_Connection, _BDTprocptr,name, property, newValue, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    Item newValue;
    Authenticator agent;
```

DESCRIPTION

The **ChangeItem()** function is used to assign a new value to an item type property. The **name** argument is comprised of three strings that identify the organization, domain, and name of an object. Wildcards may not be used to specify any portion of this argument. The **property** argument identifies the item type property for which a new value is to be assigned. The **newValue** argument is the intended new value of the property. The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **ChangeItemResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full path name of the object whose item type property was modified.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

AddItemProperty(), RetrieveItem(), ListProperties(), IsMember()

Clearinghouse2_CreateAlias, _DeleteAlias, _ListAliases

NAME

CreateAlias - add an alias to an object
DeleteAlias - delete an alias of an object
ListAliases - list objects that are aliases

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

CreateAliasResults
CreateAlias(_Connection, _BDTprocptr, alias, sameAs, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName alias;
    ThreePartName sameAs;
    Authenticator agent;


DeleteAliasResults
DeleteAlias(_Connection, _BDTprocptr, alias, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName alias;
    Authenticator agent;


void
ListAliases(_Connection, _BDTprocptr, pattern, list, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    BulkData1_Descriptor list;
    Authenticator agent;
```

DESCRIPTION

The **CreateAlias()** function is used to add a new alias to an object in the Clearinghouse database. If the object being aliased is itself an alias, the existing alias will be dereferenced before proceeding. The resulting alias will point to the actual object rather than the alias of the object. Cross-domain aliases are allowed.

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

The **ListAliases()** function is used to list the objects in a specific domain which are aliases and match **pattern**.

The **alias** argument is the name by which the object may be referenced. In the case of **CreateAlias()**, the **alias** argument is the name of the new alias to be attributed to the object. Wildcard characters may not be used.

The **sameAs** argument is the actual name, or existing alias, of the object to which the new alias will point. No wildcards may be used in specifying the **sameAs** argument.

The value of the **pattern** argument is a structure whose members specifies the **organization**, **domain**, and **object** name of the object whose aliases are to be listed. Wildcards may be used in specifying the object, but not the domain and organization. The search for an object stops upon the first occurrence of a match.

The value of the **list** argument specifies the sink that is to receive the list of aliases, in accordance to the Bulk Data Transfer Protocol. The list of aliases placed in the sink will be of type **SegmentOfObject**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

CreateAlias() returns a structure called **CreateAliasResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object to which the aliases point. **DeleteAlias()** returns a structure called **DeleteAliasResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object to which the aliases pointed. **ListAliases()** returns **void**.

ERRORS

CreateAlias() and **DeleteAlias()** both report [ArgumentError[problem], AuthenticationError[problem], CallError[problem], UpdateError[problem], WrongServer]. **ListAliases()** reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2__CreateObject

NAME

CreateObject – create a Clearinghouse object

SYNOPSIS

```
#include <courier/Clearing2__de.h>
#include <courier/except.h>

void
CreateObject(__Connection, __BDTprocptr, name, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    Authenticator agent;
```

DESCRIPTION

The **CreateObject()** function is used to create a new distinguished object in the Clearinghouse database. Distinguished means the object is not aliased.

The **name** argument is a string that specifies the object's name, domain, and organization. It may not contain wildcards.

The value of the **agent** argument is a structure whose two members contain the client's credentials and verifier, as defined in the Authentication protocol.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], UpdateError[problem], WrongServer]

SEE ALSO

CheckSimpleCredentials()

Clearinghouse2__DeleteMember, __DeleteSelf

NAME

DeleteMember - remove a member of a group type property
DeleteSelf - remove a user from a group type property

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

DeleteMemberResults
DeleteMember(__Connection, __BDTprocptr, name, property, member, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    ThreePartName member;
    Authenticator agent;


DeleteSelfResults
DeleteSelf(__Connection, __BDTprocptr, name, property, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    Authenticator agent;
```

DESCRIPTION

The **DeleteMember()** function is used to delete a member from a group type property of an object. The **DeleteSelf()** function deletes the user identified by the **agent** argument from a group type property of an object.

The **name** argument specifies the object from which the member or user is to be deleted. It is of type **ThreePartName**. Its three members, **organization**, **domain**, and **object**, identify the object in question. UNIX wildcards may not be used to specify any part of the name. If the object name encountered is an alias, it is dereferenced before it is processed.

The **property** argument indicates the group type property from which the specified member or user will be deleted.

In the case of **DeleteMember()**, the **member** argument is the name of the member that is to be deleted from the Clearinghouse database. Like the **name** argument, it is of type **ThreePartName**. UNIX wildcards may not be used to specify any part of the member. However, members of type *group* may be specified patterns, in which case, wildcards may be included in the name string and will be interpreted literally. Since the specified member is not verified by the Clearinghouse, any properly formed member name may be specified.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

DeleteMember() and **DeleteSelf()** returns structures called **DeleteMemberResults** and **DeleteSelfResults**, respectively. They both contain one member, **distinguishedObject**. It is of type **ThreePartName**. It is the full path name of the object from whose group type property the member was removed.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

AddMember(), AddSelf()

Clearinghouse2_DeleteObject

NAME

DeleteObject – delete a Clearinghouse object

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

void
DeleteObject(_Connection, _BDTprocptr, name, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName name;
    Authenticator agent;
```

DESCRIPTION

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

The **name** argument is of type **ThreePartName**, a string that specifies the object's name, domain and organization. If the **name** argument is an alias, it is first dereferenced. As a result, all aliases that point to the specified object will also be deleted. **name** may not contain any wildcard characters.

The value of the **agent** argument is a structure whose members contain the client's credentials and verifier.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], UpdateError[problem], WrongServer]

Clearinghouse2_DeleteProperty

NAME

DeleteProperty -- remove an object property

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

DeletePropertyResults
DeleteProperty(_Connection, _BDTprocptr, name, property, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName name;
    LongCardinal property;
    Authenticator agent;
```

DESCRIPTION

The **DeleteProperty()** function is used to remove a specific property from an object. Both the property number and its value are deleted. The property number may then be used again when adding new properties to the object. Note that an object is not automatically removed when its last property has been deleted.

The **name** argument is the complete Clearinghouse name of an object from which a property is to be removed. Wildcard characters may not be used to specify any portion of this argument. Aliases may be used.

The **property** argument is an integer that identifies the property to be deleted.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **DeletePropertyResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the path name of the object from which the specified property was removed.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], UpdateError[problem], WrongServer]

SEE ALSO

AddItemProperty(), AddGroupProperty(), ListProperties()

Clearinghouse2__IsMember

NAME

IsMember - determine membership of an object

SYNOPSIS

```
#include <courier/Clearing2__de.h>
#include <courier/except.h>

IsMemberResults
IsMember(__Connection, __BDTprocptr, memberOf, property, secondaryProperty, name, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName memberOf;
    LongCardinal property;
    LongCardinal secondaryProperty;
    ThreePartName name;
    Authenticator agent;
```

DESCRIPTION

The **IsMember()** function determines if a named object is a member of a group type property. **IsMember()** has two modes of operation which are determined by the **secondaryProperty** argument: normal and group. The normal mode examines only the members of a specified property. The group mode extends the search to include the membership of groups listed within the initial group property.

In normal mode, the specified member object is compared against every object belonging to a specific group type property. This process continues until the first occurrence of a match.

A database object may have numerous group or item properties associated with it. A group property contains objects which may, in turn, also contain group properties. In group mode operation, the search algorithm is such that the first group property entry encountered is examined to determine if it is an object. If it is not an object, the search algorithm continues to the next entry of the group property. If the group property entry is an object, the algorithm compares the object name against the name specified in the **memberOf** argument. If it does match, the search stops and the database object name is returned. If it does not match, the group property entry is examined further to determine if it may, in turn, contain objects having group properties. If a lower level object has a group property, the name of each object in the lower level group property is compared against the name specified in the *memberOf* argument. If there is more than one lower level group property, the algorithm searches each lower level group property for an object name that matches the one specified in the **memberOf** argument. This applies only when the group properties have a PID = **secondaryProperty**. If there are no matches, the algorithm pops back up a level to the original group property. This search algorithm is performed on every object within the original group property, including all sub-levels, until a match is found.

The **memberOf** argument is of type **ThreePartName**. It three members, **organization**, **domain**, and **object**, identify the group property to be examined. UNIX wildcards may be used in both the normal and group modes to specify any part of the **name** argument. However, wildcards will only be interpreted as such in the object name field. Wildcards used in the domain and organization fields will be interpreted as normal characters, devoid of any wildcard significance.

The **property** argument is an integer that identifies the group property to be searched.

The **secondaryProperty** argument controls the mode of operation. A value of **nullProperty**, 37777777777B, indicates that only the members of **property** are examined for **name**. If any other property ID number is entered as the value of **secondaryProperty**, then the group property having the specified ID number is also searched for the named object.

The **name** argument is of type **ThreePartName**. Its three members, **organization**, **domain**, and **object**, identify the object for whom membership is being determined. UNIX wildcards may be used but are interpreted literally. **name** may be an alias. It is not de-referenced before testing for membership.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **IsMemberResults**. It has two members: **isMember** and **distinguishedObject**. **IsMember** is a Boolean whose value indicates if the named object had been found (**TRUE**) or not (**FALSE**). **distinguishedObject** is the full path name of the object specified in the **memberOf** argument. It is of type **ThreePartName**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], WrongServer]

SEE ALSO

AddMember(), AddSelf()

Clearinghouse2_ListAliasesOf

NAME

ListAliasesOf - list the aliases of an object

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

ListAliasesOfResults
ListAliasesOf(_Connection, _BDTprocptr, pattern, list, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    BulkData1_Descriptor list;
    Authenticator agent;
```

DESCRIPTION

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

The **pattern** argument is of type **ThreePartName**, a structure whose members specify the desired **organization**, **domain**, and **object** name. UNIX wildcards may be used to specify the object, but not the domain and organization. The search for an object using wildcards stops upon the first occurrence of a match. If the object name encountered is an alias, it is dereferenced before its aliases are determined.

The **list** argument specifies the sink that is to receive the aliases of an object, in accordance to the Bulk Data Transfer Protocol. The list of aliases placed in the sink will be of type **SegmentOfObjectName**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **ListAliasesOfResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object to which the aliases point.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2_ListDomain

NAME

ListDomain - list the domains in an organization

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

void
ListDomain(_Connection, _BDTprocptr, pattern, list, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    BulkData1_Descriptor list;
    Authenticator agent;
```

DESCRIPTION

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

The **pattern** argument is a text string that specifies the target organization and domain(s). UNIX wildcards may be used to specify the domain, but not the organization. The search continues through the entire Clearinghouse database, returning all the domain names that match the specified pattern. The **list** argument specifies the sink that is to receive the list of organizations, in accordance to the Bulk Data Transfer Protocol. The list of domains placed in the sink will be of type **SegmentOfDomain**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2_ListDomainServed

NAME

ListDomainServed - determine the domains served by a Clearinghouse

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

void
ListDomainServed(_Connection, _BDTprocptr, domains, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    BulkData1_Descriptor domains;
    Authenticator agent;
```

DESCRIPTION

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

The **domains** argument is a bulk data transfer parameter that specifies the sink that is to receive the list of domains in accordance with the Bulk Data Transfer Protocol. The data returned to the sink is of type **SegmentOfDomainName**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns **void**.

ERRORS

Reports AuthenticationError[problem], CallError[problem]]

Clearinghouse2_ListObjects

NAME

ListObjects - list objects in a domain

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

void
ListObjects(_Connection, _BDTprocptr, pattern, property, list, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    LongCardinal property;
    BulkData1_Descriptor list;
    Authenticator agent;
```

DESCRIPTION

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

The **pattern** argument is of type **ThreePartName**, a structure whose members specify the desired **organization**, **domain**, and **object** names. UNIX wildcards may be used to specify the object, but not the domain and organization. The **property** argument is used to specify a property that each object matching the search pattern must have in order for it to be listed. One property number of particular importance is 0. 0 is synonymous with **all**. When 0 is specified, it indicates that all the objects in a domain that match the pattern, regardless of intrinsic properties, are to be listed.

The **list** argument specifies the sink that is to receive the list of domains, in accordance to the Bulk Data Transfer Protocol. The list of organizations placed in the sink will be of type **SegmentOfObject**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2__ListOrganizations

NAME

ListOrganizations – list Clearinghouse organizations

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

void
ListOrganizations(_Connection, _BDTprocptr, pattern, list, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    String pattern;
    BulkData1_Descriptor list;
    Authenticator agent;
```

DESCRIPTION

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

The **pattern** argument is a string that specifies the set of organizations to be listed. **pattern** is typically the partial spelling of the desired Clearinghouse organization names . Wildcard characters may be included in the partial spelling. The search continues through the entire Clearinghouse database, returning all organization names that match the specified pattern. The **list** argument specifies the sink that is to receive the list of organizations, in accordance to the Bulk Data Transfer Protocol. The list of organizations placed in the sink will be of type **SegmentOfOrganization**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2_ListProperties

NAME

ListProperties -- list the property numbers of an object

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

ListPropertiesResults
ListProperties(_Connection, _BDTprocptr, pattern, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    Authenticator agent;
```

DESCRIPTION

The **ListProperties()** function is used to list the ID number of every property associated with an object. The **pattern** argument is of type **ThreePartName.** It is a structure whose members specify the **organization**, **domain**, and **object** name of the object whose property numbers are to be listed. UNIX wildcards may be used in specifying the object, but not the domain or organization. The search for an object using wildcards stops upon the first occurrence of a match.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **ListPropertiesResults**. Its has two members: **distinguishedObject** and **properties. distinguishedObject** is of type **ThreePartName**. It is the full name of the object in question. **properties** is of type **Properties**. **properties** is a list of the properties associated with the object. Note that properties are referred to by number, not name.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

SEE ALSO

AddItemProperty(), AddGroupProperty()

Clearinghouse2__LookupObject

NAME

LookupObject – retrieve an object name

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

LookupObjectResults
LookupObject(_Connection, _BDTprocptr, name, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName name;
    Authenticator agent;
```

DESCRIPTION

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

The **name** argument is the name of the object in the Clearinghouse. The name that is specified may be a partial spelling, an alias, or both. Wildcard characters may be included in the partial spelling of the object name, but not the domain and organization. The search continues until the first occurrence of the named object, or its alias, is encountered. If the object's alias is encountered, it is dereferenced before being returned to the calling function.

The value of the **agent** argument is a structure whose members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **LookupObjectResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the complete name of the Clearinghouse database object in question.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], WrongServer]

Clearinghouse2_RetrieveAddresses

NAME

RetrieveAddresses - query a server for its network addresses

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

RetrieveAddressesResults
RetrieveAddresses(_Connection, _BDTprocptr)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
```

DESCRIPTION

The **RetrieveAddresses()** function is used to query the clearinghouse server for a list of all of its network addresses. This function knows the Clearinghouse server to access based upon the value of the **_Connection** argument. This function may also be used as a check to insure the Clearinghouse server is available before calling other functions.

RETURN VALUE

This function returns a structure called **RetrieveAddressesResults**. Its one member, **address**, is of type **NetworkAddressList**. It contains a list of the network addresses recognized by the Clearinghouse. A network address entry is defined in Xerox Network Systems Architecture as host number (48b:1), network number (32 bit integer), and a socket number (16 bit integer). The maximum number of entries returned is 40.

ERRORS

Reports [CallError[problem]]

Clearinghouse2_RetrieveItem

NAME

RetrieveItem - list the value of an item type property

SYNOPSIS

```
#include <courier/Clearing2_de.h>
#include <courier/except.h>

RetrieveItemResults
RetrieveItem(_Connection, _BDTprocptr, pattern, property, agent)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName pattern;
    LongCardinal property;
    Authenticator agent;
```

DESCRIPTION

The **RetrieveItem()** function is used to determine the value of an item type property that is associated with an object. This function returns both the distinguished object name, and the value of the item property.

The **pattern** argument is of type **ThreePartName**, a structure whose members specify the **organization**, **domain**, and **object** name of the object from which the value of **property** is to be extracted. UNIX wildcards may be used in specifying the object, but not the domain and organization. The search for an object using wildcards stops upon the first occurrence of a match. If the object name encountered is an alias, it is dereferenced before it is returned.

The **property** argument specifies the ID number of the property for which its value is to be returned. UNIX wildcards may not be used.

Properties are referred to by ID number, not name. One property number of particular importance is 0. 0 is synonymous with **all**. When 0 is specified, it indicates that all the item properties of the first object encountered that matches the specified **pattern** are to be returned.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **RetrieveItemResults**. It has two members: **distinguishedObject** and **value**. **distinguishedObject** is of type **ThreePartName**. It is the full name of the object whose item type property is being listed. **value** is of type **Item**. It contains the value of the item property.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem],
PropertyError[problem], WrongServer]

SEE ALSO

AddItemProperty(), AddGroupProperty(), ListProperties(), IsMember()

Clearinghouse2__RetrieveMembers

NAME

RetrieveMembers - retrieve the value of a group type property

SYNOPSIS

```
#include <courier/Clearing2__de.h>
#include <courier/except.h>

RetrieveMembersResults
RetrieveMembers(__Connection, __BDTprocptr, pattern, property, membership, agent)
    COURIERFD __Connection;
    int (*__BDTprocptr)();
    ThreePartName pattern;
    LongCardinal property;
    BulkData1__Descriptor membership;
    Authenticator agent;
```

DESCRIPTION

The **RetrieveMembers()** function is used to extract, or retrieve, the value of a group type property associated with an object. The **pattern** argument is of type **ThreePartName**. It is a structure whose members, **organization**, **domain**, and **object**, identify the object in question. UNIX wildcards may be used to specify the object, but not the domain and organization. The search for an object using wildcards stops upon the first occurrence of a match. If the object name encountered is an alias, it is dereferenced.

The **property** argument identifies the property from which a value is to be retrieved. One property number of particular importance is 0. 0 is synonymous with **all**. When 0 is specified, it indicates that all the group properties that match the criteria specified in **pattern** are to be returned.

The **membership** argument is a bulk data parameter that specifies the sink that is to receive the list of values in accordance to the Bulk Data Transfer Protocol. The data received via the membership argument is of the type **StreamOfThreePartName**.

The **agent** argument is a structure whose two members contain the client's credentials and verifier.

RETURN VALUE

This function returns a structure called **RetrieveMembersResults**. Its one member, **distinguishedObject**, is of type **ThreePartName**. It is the full name of the object from which the property value was extracted.

ERRORS

Reports [ArgumentError[problem], AuthenticationError[problem], CallError[problem], PropertyError[problem], WrongServer]

SEE ALSO

RetrieveItem(), IsMember()

Filing6_Close

NAME

Close - terminate a file handle

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
Close(_Connection, _BDTprocptr, file, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    Session session;
```

DESCRIPTION

The **Close()** function is used to indicate to the File Service that a specific file handle is no longer wanted for the remainder of the current session. The File Service then releases acquired resources, such as locks associated with the handle, and invalidates the file handle. If no other file handle is associated with it, the file buffer is also purged.

The **file** argument is the file handle originally returned by a call to the **Open()** function. It specifies the file that is to be closed. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], HandleError, SessionError[problem], UndefinedError]

SEE ALSO

Open(), Logon()

Filing6_Continue

NAME

Continue - lengthen the duration of an inactive session

SYNOPSIS

```
#include <courier/Filing6  de.h>
#include <courier/except.h>

ContinueResults
Continue(  Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (* _BDTprocptr)();
    Session session;
```

DESCRIPTION

The **Continue()** function is used to determine the duration, in seconds, permitted an inactive session before it is terminated by the File Service. The duration of inactivity permitted a session is determined by the File Service. A call to **Continue()**, as with all other remote function calls, is considered activity and therefore, the session is reallocated the full amount of time permitted an inactive session. The **session** argument is the session handle returned by an call to **Logon()**. It is the session to be lengthened.

RETURN VALUE

This function returns a structure called **ContinueResults**. Its one member, **continuance**, is a cardinal number that specifies the timeout period of the file server. The timeout is specified in units of seconds. The returned value indicates the frequency with which the client must perform some activity. For example, to determine the timeout period of a session, use Continue():

Continue(token, (11B,27734B), verifier**)**

It returns:

(600)

Therefore, to prevent termination of the current session some activity must occur within every ten minutes (600 seconds).

ERRORS

Reports [AuthenticationError[problem], SessionError[problem], UndefinedError]

SEE ALSO

Logon()

Filing6_Copy

NAME

Copy - create a duplicate file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

CopyResults
Copy(_Connection, _BDTprocptr, file, destinationDirectory, attributes, controls, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    Handle destinationDirectory;
    AttributeSequence attributes;
    ControlSequence controls;
    Session session;
```

DESCRIPTION

The **Copy()** function is used to duplicate an existing file or directory. If the object to be copied is a directory, all the descendants are also copied. The duplicate file or directory is then placed into a specified directory. A file or directory cannot be copied into itself or any of its descendants.

The **file** argument is the file handle of the file or directory to be copied. Read access (i.e., read permission) is required of the file. If the object is a directory, then read access is required of all its descendants.

The **destinationDirectory** argument is the file handle for the directory in which the copy is to be placed. Add access (i.e.,write permission) is required of the destination directory. The value of **destinationDirectory** may be set to **nullHandle**, thus indicating that the resulting file is to be placed in the root directory.

The **attributes** argument specifies the sequence of characteristics to be assigned to the new file or directory, thus overriding those of the original file or directory.

The **controls** argument specifies the access permissions of the new file or directory.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns a structure called **CopyResults**. Its one member, **newFile**, is of type **Handle**. It is a handle for the newly created file or directory.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], HandleError, SessionError[problem], UndefinedError]

Filing6_Create

NAME

Create - make a new file

SYNOPSIS

```
#include <courier/ Filing6 de.h>
#include <courier/except.h>

CreateResults
Create(_Connection, _BDTprocptr, directory, attributes, controls, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle directory;
    AttributesSequence attributes;
    ControlSequence controls;
    Session session;
```

DESCRIPTION

The **Create()** function is used to make a new file. **Create()** is particularly useful for creating directories. If the file to be created is also to contain some data, use the **Store()** function.

The **directory** argument is the file handle for the directory in which the created file will be placed. **nullHandle** may be specified, indicating that the file will be placed in the root level directory.

attributes are data items associated with a file. See **Open()** for a description of **attributes**.

controls define the nature of permissible file access that a file handle gives to the client. **controls** may be specified in any function that returns a file handle. The **controls** specified apply only to the returned handle. **controls** is a structure that contains three enumerated types: **LOCK**, **TIMEOUT**, and **ACCESS**.

LOCK offers three choices: **NONE**, **SHARE**, and **EXCLUSIVE**. **NONE** indicates that there are no access restrictions. **SHARE** means that other sessions cannot move or delete the file, and cannot place an exclusive lock on the file. **EXCLUSIVE** means that other sessions cannot move or delete the file, and cannot place a **SHARE** or **EXCLUSIVE** lock on the file.

TIMEOUT is an integer that indicates the number of seconds that the File Service will wait after a client requests a lock on a file that is unavailable. If the time specified is exceeded and the locked file does not become available, an error is returned. The interval that the File Service will wait is usually an implementation-dependent constant, though you may specify an overriding interval. If a **TIMEOUT** of zero is specified, the File Service will not wait. If the locked file is unavailable, an error is immediately returned. If 177777B (**defaultTimeout**) is specified, the implementation-dependent default is applied. If no **TIMEOUT** is specified, **defaultTimeout** is assumed.

ACCESS specifies the operations permitted a particular file handle with respect to a file or its children. If access permissions have not been enabled, the file handle may not be used in any operation that attempts to access the specific file. The six acceptable values of access are: **READ**, **WRITE**, **OWNER**, **ADD**, **REMOVE**, and **FULLACCESS**(177777B).

READ means the client may read the contents and attributes of a file. If it is a directory, the client may enumerate its children and search for files in that directory. **WRITE** permits the client to modify the contents and attributes of the file. This includes deleting the file. If it is a directory, a client may also change environment attributes access lists of the directory's children. **OWNER** means a client may change the file's access list. **ADD** permits a client to add subdirectories and files. **REMOVE** may only be applied to directories. It allows a client to delete subdirectories. **FULLACCESS**(177777B) means a client is granted the complete set of access permissions. That is, read, write, change the access list (owner), add, and remove. **FULLACCESS** cannot be specified along with any of the preceding access types.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns a structure called **CreateResults**. Its one member, **file**, is of type **Handle**. It contains the file handle for the newly created file.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ControlTypeError, ControlValueError, HandleError, InsertionError, SessionError[problem], SpaceError, UndefinedError]

Filing6_Delete

NAME

Delete - remove an existing file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
Delete(_Connection, _BDTprocptr, file, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    Session session;
```

DESCRIPTION

The **Delete()** function is used to remove an existing file. When this function is called, the target file is closed and then deleted. All resources allocated to the file are then freed for other uses. Once the file is deleted, the file handle associated with it becomes invalid.

The **file** argument is the file handle of the file to be deleted. The file to be deleted can have only one file handle during the current session. If the **file** handle specifies a directory, the directory and all its descendents will be deleted.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], HandleError, SessionError[problem], UndefinedError]

SEE ALSO

Logon()

Filing6_Find

NAME

Find - locate a file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

FindResults
Find(_Connection, _BDTprocptr, directory, scope, controls, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle directory;
    ScopeSequence scope
    ControlSequence controls;
    Session session;
```

DESCRIPTION

The **Find()** function is used to locate and open a file in a directory. The File Service enumerates the directory's descendants in accordance to the ordering attribute of the directory. The first file that meets the search criteria is opened. If no file matches the search criteria, an error is reported.

The **directory** argument is the file handle of the directory whose descendants are to be enumerated. **nullHandle** may be specified to indicate that the search is to begin at the root directory. Read access (i.e., read permission) is required of the directory to be enumerated. The **scope** argument specifies a criteria by which to search for files.

The **scope** argument is a structure comprised of **COUNT**, **DIRECTION**, **FILTER**, and **DEPTH**.

> **COUNT** specifies the maximum number of files to be viewed by the client. The **unlimitedCount** constant may be specified as the value of **COUNT** to indicate that there is no limit to the files to be viewed.
>
> **DIRECTION** specifies the order in which files are enumerated. **DIRECTION** is used by those functions that list (display files in a specified direction) or search (display files that match a specific criteria). **DIRECTION** an enumerated type that accepts one of two values: **FORWARD** or **BACKWARD**. A value of **FORWARD** indicates that enumeration is to begin with the first file in the sequence of ordered files and end with the last. A value of **BACKWARD** indicates that enumeration is to begin with the last file in the sequence and end with the first. If **DIRECTION** is not specified, a **FORWARD** direction is assumed.
>
> **FILTER** is a set of Boolean operators and special characters that assist in differentiating files of interest.
>
> **DEPTH** is an integer that specifies the maximum number of levels down the directory hierarchy in which to search for files. A value of **1** indicates that only the files in the specified directory are to be considered. A value of **2** indicates that the directories immediately below the specified directory are also to be considered when searching for files. The **allDescendants** constant may be specified as the value of **DEPTH** to indicate that there is no restriction on the levels of directory hierarchy to descend. That is, all directories below the specified directory will be considered when searching for files. If **DEPTH** is not specified, a **DEPTH** of **1** is assumed.

The **controls** argument specifies the access permissions to be applied to the new file handle.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns a structure called **FindResults**. Its one member is **handle** is of type **Handle**. It contains the file handle of the first file that matches the search criteria.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], ControlTypeError, ControlValueError, HandleError, ScopeTypeError, ScopeValueError, SessionError[problem], UndefinedError]

Filing6_GetAttributes, _ChangeAttributes

NAME

GetAttributes - retrieve the attributes of a file
ChangeAttributes - modify file attributes

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

GetAttributesResults
GetAttributes (_Connection, _BDTprocptr, file, types, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    AttributeTypeSequence types;
    Session session;

void
ChangeAttributes (_Connection, _BDTprocptr, file, attributes, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    AttributeSequence attributes;
    Session session;
```

DESCRIPTION

The **GetAttributes()** function is used to retrieve the attribute and attribute value pairs of a specific file. When this function is called, the File Service attempts to obtain the requested attributes and returns them to the requester. The **ChangeAttributes()** function is used to modify the access-related attributes of a specific file.

The **file** argument is the file handle of the file whose attributes are to be retrieved or changed. Depending upon the changes to be made, you must have appropriate access permission. Write access is required if only data attributes are to be changed. Write access to the file's parent is required for environment-related attribute changes. Write access to the file's parent or owner access to the file itself is required if accessList or defaultAccessList attributes are to be changed. Changes made to a file's access list attributes takes immediate effect. All handles to the file within the current session and all new handles acquired later are affected. Access list changes made in the current session may not affect the existing sessions of other clients until those sessions are terminated.

The **types** argument is a sequence of types for which the values are to be returned. The **allAttributeTypes** constant is a cardinal number that may be specified as the value of the **attributes** argument to retrieve all the attributes of the file. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

The **attributes** argument is a sequence of the attributes to be changed.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

GetAttributes() returns a structure called **GetAttributesResults**. Its one member, **attributes**, is of type **AttributeSequence**. It is a sequence of attributes that corresponds one-to-one with the items specified in the **types** argument. **ChangeAttributes()** returns **void**.

ERRORS

GetAttributes() reports [AccessError[problem], AttributeTypeError, AuthenticationError[problem], HandleError, SessionError[problem], UndefinedError] **ChangeAttributes()** reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], HandleError, InsertionError, SessionError[problem], SpaceError, UndefinedError]

SEE ALSO

Create(), Logon(), Open()

Filing6_GetControls, _ChangeControls

NAME

GetControls - return the controls associated with a specific file
ChangeControls - modify the controls associated with a specific file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

GetControlsResults
GetControls(_Connection, _BDTprocptr, file, types, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    ControlTypeSequence types;
    Session session;

void
ChangeControls(_Connection, _BDTprocptr, file, controls, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    ControlSequence controls;
    Session session;
```

DESCRIPTION

The **GetControls()** function is used to determine the file access associated with a specific file. Only the values of the specified controls will be returned. The **ChangeControls()** function is used to modify specific controls associated with a file. If a lock is specified, the File Service will attempt to acquire it, and if successful, any prior lock is released. Refer to **Create()** for more information regarding controls.

The **file** argument is the file handle of the file from which to extract or change control values. The **types** argument is a sequence of integers that indicates the specific controls for which you are attempting to retrieve the values. The **controls** argument is a sequence of the control items to be reset. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

GetControls() returns a structure called **GetControlsResults**. Its one member, **controls**, is of type **ControlSequence**. It is a sequence of control items that corresponds one-to-one with the items specified in the **types** argument. **ChangeControls()** returns **void**.

ERRORS

GetControls() reports [AccessError[problem], AuthenticationError[problem], ControlTypeError, HandleError, SessionError[problem], UndefinedError]. **ChangeControls()** reports [AccessError[problem], AuthenticationError[problem], ControlTypeError, ControlValueError, HandleError, SessionError[problem], UndefinedError]

SEE ALSO

Create()

Filing6_List

NAME

List - display the files in a directory

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
List(_Connection, _BDTprocptr, directory, types, scope, listing, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle directory;
    AttributeTypeSequence types;
    ScopeSequence scope;
    BulkData1_Descriptor listing;
    Session session;
```

DESCRIPTION

The **List()** function is used to enumerate the files in a directory and return desired attributes. The File Service enumerates the directory in accordance to its ordering attribute. The requested attributes of those files meeting desired criteria specified in the **scope** argument are returned. Since attributes are obtained with varying degrees of difficulty, it is recommended that you only request necessary attributes.

The files in the directory may change at any time while this function is being executed. Therefore, it is possible that the set of files returned may not reflect the directory in its current state. If a depth greater than 1 is specified, then descendants of the specified directory must also be considered. To prevent changes from invalidating the results of **List()**, it necessary to acquire a **SHARE** lock on the directory before calling the **List()** function.

The **directory** argument is of type **Handle**,. It is the file handle for the directory to be enumerated. **nullHandle** may be specified to indicate that enumeration is to begin with the root directory. Read access (i.e., read permission) is required of the directory to be enumerated. This also includes all the subdirectories to be enumerated. The **types** argument specifies a sequence of attributes a file must have to be considered. The **allAttributeTypes** constant may be specified as the value of **types** to indicate that all files are to be considered, regardless of the attributes they posses.

The **scope** argument specifies a criteria by which to search for files. The **scope** argument is a structure comprised of **COUNT**, **DIRECTION**, **FILTER**, and **DEPTH**. See the description of **scope** in **Find()** for more information.

The **listing** argument specifies the sink that is to receive the data in accordance to the Bulk Data Transfer Protocol. The transferred bulk data is of type **StreamOfAttributeSequence**.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AuthenticationError[problem], ConnectionError, HandleError, ScopeTypeError, ScopeValueError, SessionError[problem], TransferError[problem], UndefinedError]

SEE ALSO

Find(), Open(), Logon()

Filing6__Logoff

NAME

Logoff - end a current File Service session

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
Logoff(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **Logoff()** function is used to end the current File Service session. Upon calling this function, the File Service verifies that the request is valid, terminates the session, releases any allocated resources, and then invalidates the session handle. The **session** argument is the session handle returned by a call to **Logon()**.

RETURN VALUE

This function returns void.

ERRORS

Reports [AuthenticationError[problem], ServiceError[problem], SessionError[problem], UndefinedError]

SEE ALSO

Logon()

Filing6_Logon

NAME

Logon - begin a File Service session

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>
#include <courier/FIlingSu1.h>

LogonResults
Logon( _Connection, _BDTprocptr, service, credentials, verifier)
    COURIERFD _Connection;
    int (* _BDTprocptr)();
    ThreePartName service;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **Logon()** function is used to initiate access to a File Service. This function must be executed before any other Filing Protocol function. If the File Service verifies that the Logon request is valid, it will create a session, and return a session handle. This session handle is an identifier that must accompany any other filing function call.

The **service** argument is the name of the Filing Service to be accessed. If a service is not explicitly specified, and thus the service name is left null, a default service is provided by the installed XNS system. The **credentials** and **verifier** arguments identify the client, or user, initiating a File Service session. **credentials** may be in one of several forms: primary, secondary, or encrypted secondary.The **verifier** is the simple verifier returned earlier by the Authentication Service.

RETURN VALUE

This function returns a pointer to a structure, called **LogonResults**. This structure is similar to the **session** structure and is also referred to as the session handle. Its one member, **session**, is of type **Session**. It is a structure having two members: **token** and **verifier**. The **token** array identifies the session to the File Service, thereby identifying the user and the status of the user's interaction with the File Service. The session token, once returned, is to be used in subsequent function calls to the File Service within the same session. The token remains static for the duration of the session and it cannot be interpreted by the client. The **verifier** array is defined by the Authentication Protocol. It verifies that all function calls using the same session handle have been originated from the same client that originally established the session. The verifier is not static and may change with each new function call.

ERRORS

Reports [AuthenticationError[problem], ServiceError[problem], SessionError[problem], UndefinedError]

Filing6_Move

NAME

Move - move a file to another directory

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
Move(_Connection, _BDTprocptr, file, destinationDirectory, attributes, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    Handle destinationDirectory;
    AttributeSequence attributes;
    Session session;
```

DESCRIPTION

The **Move()** function is used to change the directory structure of the filing service without creating or deleting files. The File Service moves a file or directory to a specific directory location. If the specified file is a child of another directory, it is removed from that directory. If a temporary file is specified, it is made permanent. If the specified file has descendants, they will remain as such and will be moved along with the file. A file cannot be moved into itself or any of its descendants.

The **file** argument is the file handle of the file or directory to be moved. Read and write access (i.e., read and write permission) is required of the file or directory to be moved. Remove access is required of the file's parent directory. There can be only one file handle in use during the current session for the file specified. If there is more than one file handle in use for a file, it cannot be moved.

The **destinationDirectory** argument is the file handle for the directory in which the file is to be placed. Add access (i.e., write permission) is required of the destination directory. The **attributes** argument specifies the sequence of characteristics to be assigned to the new file or directory, thus overriding those of the original file or directory. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ControlTypeError, ControlValueError, HandleError, InsertionError, SessionError[problem], SpaceError, UndefinedError]

Filing6_Open

NAME

Open - make a file available for use

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

OpenResults
Open(_Connection, _BDTprocptr, attributes, directory, controls, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    AttributesSequence attributes;
    Handle directory;
    ControlSequence controls;
    Session session;
```

DESCRIPTION

The **Open()** function is used to make a file available for use. Once this function is called, the file server prepares the specified file for use, applies specific controls to it, then creates and returns a file handle. The file is marked "in use" and attempts to move or delete it by other sessions is ignored.

The **attributes** argument identifies the file to be opened. It requires six parameters: **parentID**, **fileID**, **name**, **pathname**, **type**, and **version**. **parentID** specifies the starting directory in which may be found a file containing the same ID as that specified in the **fileID** parameter. If **parentID** is omitted, the starting directory is the root directory.

fileID identifies the file that is to be opened. If **parentID** or directory is included in the function call, the specified file must be a child of the starting directory. If neither of the two is specified, the file may be anywhere. The **name** parameter supplies the name of the file to be opened. The file specified in this parameter must be a child of the starting directory. The **pathname** parameter specifies the path name of the file to be opened. The first component of **pathname** must be a child of the starting directory. If the starting directory is omitted, the root directory is used. The client must have the appropriate access permissions for every file specified in the path name. The **type** parameter indicates the file type of the object to be opened. The version parameter specifies the version number of the file to be opened. If the parameter is omitted, the file with the highest version number is opened. This parameter is ignored if the last file named in the pathname argument explicitly states the version number. This parameter is specified only if the **name** parameter or **pathname** parameter is used.

The sequence for specifying the attributes are as follows. The brackets indicate optional parameters:

a) **fileID [parentID] [type]**
b) **name [parentID] [type] [version]**
c) **pathname [parentID] [type] [version]**

The **directory** argument specifies a starting directory from which to begin the search for the file specified in the attributes argument. **nullHandle** may be specified in the **directory** argument rather than a valid session handle. **nullHandle** is a reserved constant with special significance. It may be used in functions, like **Open()**, to imply the root directory. Unless specifically stated, a **nullHandle** is not to be used as an argument value.

RETURN VALUE

This function returns a structure called **OpenResults**. Its one member, **handle**, is of type **Handle**. It is the file handle for the file identified by the **attributes** argument. It is to be passed as an argument to all further calls to functions that are to access the file during the current session.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ControlTypeError, ControlValueError, HandleError, SessionError[problem], UndefinedError]

SEE ALSO

Logon()

Filing6_Replace

NAME

Replace - replace the contents of a file

SYNOPSIS

```
#include <courier/Filing6 de.h>
#include <courier/except.h>

void
Replace( Connection, _BDTprocptr, file, attributes, content, session)
    COURIERFD _Connection;
    int (* _BDTprocptr)();
    Handle file;
    AttributeSequence attributes;
    BulkData1_Descriptor content;
    Session session;
```

DESCRIPTION

The **Replace()** function is used to remove the contents of a file and then replace it with data received from a specific source.

The **file** argument is the file handle for the file whose contents is to be replaced. Write access (i.e., write permission) is required of the specified file. The **attributes** argument specifies the sequence of characteristics to be assigned to the resulting file. The **content** argument specifies the source that is to supply the data to go in the replacement file in accordance with the Bulk Data Transfer Protocol. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ConnectionError, HandleError, SessionError[problem], SpaceError, TransferError[problem], UndefinedError]

Filing6_Retrieve

NAME

Retrieve - extract the contents of a file

SYNOPSIS

```
#include <courier/ Filing6  de.h>
#include  <courier/except.h>

void
Retrieve(  Connection, _BDTprocptr, file, content, session)
    COURIERFD _Connection;
    int (* _BDTprocptr)();
    Handle file;
    BulkData1 _Descriptor content;
    Session session;
```

DESCRIPTION

The **Retrieve()** function is used to read the contents of an existing file and transfer them to the client.

The **file** argument is the file handle of the file from which the contents are to be retrieved. Read access (i.e., read permission) is required of the specified file. The **content** argument specifies the sink that is to receive the contents of the file in accordance with the Bulk Data Transfer Protocol. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], ConnectionError, HandleError, SessionError[problem], TransferError[problem], UndefinedError]

Filing6_RetrieveBytes, _ReplaceBytes

NAME

RetrieveBytes - read bytes within a file
ReplaceBytes - modify the contents of a file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
RetrieveBytes(_Connection, _BDTprocptr, file, range, sink, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    ByteRange range;
    BulkData1_Descriptor sink;
    Session session;

void
ReplaceBytes(_Connection, _BDTprocptr, file, range, source, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    ByteRange range;
    BulkData1_Descriptor source;
    Session session;
```

DESCRIPTION

The **RetrieveBytes()** function is used to read a range of bytes within a file. The **ReplaceBytes()** function is used to overwrite the contents of a file or to append new data to a file.

The **file** argument is the file handle for the file from which to retrieve a range of bytes or modify a range of bytes. Read access (i.e., read permission) is required of the file from which you want to retrieve data. Write access (i.e., write permission) is required of the file to be modified.

When calling **RetrieveBytes()**, the **range** argument is of type **ByteRange**. It specifies the contiguous sequence of bytes to be returned. When calling **ReplaceBytes()**, the **range** argument specifies the file location, in bytes, where data is to be inserted and the total number of bytes to be inserted. The value of the **range** argument and the data supplied by the source must be the same. If the **firstByte** parameter of the **range** argument is set to **endOfFile**, the supplied data is appended to the specified file. Otherwise, the supplied data replaces the file data that starts at the specified file location, ending at however many bytes are specified for the length. In the case of appending data, this function insures that all the data is successfully appended or it will not modify the file at all.

> **ByteRange** is a structure comprised of **ByteAddress** and **ByteCount**. These two members specify the byte offset at which to begin storing or retrieving data and the number of bytes to store or retrieve, respectively. **ByteAddress** is a LongCardinal number. The value specified cannot exceed the total size in bytes of the file. Call the **GetAttributes()** function with the **dataSize** argument to ascertain the total size in bytes of the file. **ByteCount** is a LongCardinal number that indicates the total number of contiguous bytes to store or retrieve. The value specified, when added to the offset, cannot exceed the total size in bytes of the file.
>
> The **endOfFile** constant is a LongCardinal number that may be used as the value of **ByteAddress** or **ByteCount** to refer to the logical end of a file. As a byte address, **endOfFile** is used to refer to the byte

position at the end of a file where new data can be appended. When used as a **ByteCount**, **endOfFile** may be used to represent the number of bytes that begins at the specified offset and ends at the last byte defined for the file.

The **sink** argument specifies the sink that is to receive the requested data bytes in accordance with the Bulk Data Transfer Protocol. The **source** argument specifies the source that is to supply the data bytes in accordance with the Bulk Data Transfer Protocol.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

These functions return **void**.

ERRORS

Reports [AccessError[problem], HandleError, RangeError, SessionError[problem], UndefinedError]

SEE ALSO

GetAttributes(), Open(), Logon()

Filing6_Serialize, _Deserialize

NAME

Serialize - encode a file
Deserialize - unencode a file

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
Serialize(_Connection, _BDTprocptr, file, serializedFile, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle file;
    BulkData1_Descriptor serializedFile;
    Session session;

DeserializeResults
Deserialize(_Connection, _BDTprocptr, directory, attributes, controls, serializedFile, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle directory;
    AttributeSequence attributes;
    ControlSequence controls;
    BulkData1_Descriptor serializedFile;
    Session session;
```

DESCRIPTION

The **Serialize()** function is used to compress all the descriptive information and data of a file and its descendants into a series of eight-bit bytes. The resulting data is a single object of type **SerializedFile**. This object is then transferred to a sink. It is necessary to serialize a file in order to transfer it to another File Service or store it on some other medium.

The **Deserialize()** function is used to reconstruct a file and its descendants from a previously serialized file. When this function is called, a new file is created in the specified directory and a file handle for the new file is returned. The new file will have most of the attributes, all the contents and all the descendants as it did prior to serialization. Some attributes are ignored during deserialization because the attribute duplicates information that is implicit to other data. For example, the numberOfChildren attribute is ignored because the number of descendants a file has is already encoded in the serialized file. If the name of the deserialized file duplicates that of an existing file, the deserialized file is created with an appropriate version number. The existing file is not replaced by the deserialized file.

The **file** argument is the file handle for the file whose contents is to be serialized. Read access (i.e., read permission) is required of the specified file.

The **serializedFile** argument, either, specifies the sink that is to receive the compressed file contents in the case of **Serialize()**, or specifies the source that is to supply the serialized file data in the case of **Deserialize()**. The specifications are made in accordance to the Bulk Data Transfer Protocol.

The **directory** argument is a handle of the directory in which the new file is to be placed. **directory** may be set to **nullHandle**, thus indicating that the resulting file is to be placed in the root directory. Add access (i.e., write permission) is required of the destination directory if the file handle specified is not **nullHandle**. The **attributes** argument specifies the sequence of characteristics to be assigned to the new file, thus overriding the default characteristics inherent to the serialized file. The **controls** argument specifies the access permissions to be applied to the new file handle.

The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

The **Serialize** function returns **void**. The **Deserialize** function returns a structure called **DeserializeResults**. Its one member, **file**, is of type **Handle**. It is the file handle for the file identified in the **attributes** argument.

ERRORS

Serialize reports [AccessError[problem], AuthenticationError[problem], ConnectionError, HandleError, SessionError[problem], TransferError[problem], UndefinedError] **Deserialize** reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ConnectionError, ControlTypeError, ControlValueError, HandleError, InsertionError, SessionError[problem], SpaceError, TransferError[problem], UndefinedError]

SEE ALSO

Open(), Logon()

Filing6__Store

NAME

Store - create a file

SYNOPSIS

```
#include <courier/Filing6 de.h>
#include <courier/except.h>

StoreResults
Store( Connection,  BDTprocptr, directory, attributes, controls, content, session)
    COURIERFD  Connection;
    int (*  BDTprocptr)();
    Handle directory;
    AttributeSequence attributes;
    ControlSequence controls;
    BulkData1 Descriptor content;
    Session session;
```

DESCRIPTION

The **Store()** function is used to create a file that contains specific data. When this function is called, a new file is created with specific attributes and is placed in a specified directory. It is then filled with data sent by the client in accordance to the Bulk Data Transfer Protocol. Upon completion, a file handle is returned for the new file.

The **directory** argument is the file handle for the directory in which the new file is to be placed. **nullHandle** may be specified to indicate that the resulting file is to be placed in the root directory. Add access (i.e., write permission) is required of the destination directory if the file handle specified is not **nullHandle**. The **attributes** argument specifies the sequence of characteristics to be assigned to the new file. The **controls** argument specifies the access permissions of the new file. The **content** argument specifies the source that is to supply the contents of the new file in accordance with the Bulk Data Transfer Protocol. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

RETURN VALUE

This function returns a structure called **StoreResults**. Its one member, **file**, is of type **Handle**. It is the file handle for the file identified in the **attributes** argument.

ERRORS

Reports [AccessError[problem], AttributeTypeError, AttributeValueError, AuthenticationError[problem], ConnectionError, ControlTypeError, ControlValueError, HandleError, InsertionError, SessionError[problem], SpaceError, TransferError[problem], UndefinedError]

SEE ALSO

Retrieve()

Filing6_UnifyAccessLists

NAME

UnifyAccessLists - group the access lists of a subtree of files

SYNOPSIS

```
#include <courier/Filing6_de.h>
#include <courier/except.h>

void
UnifyAccessLists(_Connection, _BDTprocptr, directory, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Handle directory;
    Session session;
```

DESCRIPTION

The **UnifyAccessLists()** function is used to assign the access list attributes (i.e., permissions) of a directory to all its descendants. The **file** argument is the file handle of the directory. Write access is required of the directory specified as well as all its descendants. The **session** argument is the client's session handle that was returned upon executing the **Logon()** function.

Changes made to access list attributes takes immediate effect. All handles to the files within the current session and all new handles acquired later are affected. Access list changes made in the current session may not affect the existing sessions of other clients until those sessions are terminated.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], HandleError, SessionError[problem], UndefinedError]

SEE ALSO

Logon(), Open()

Gap3_Create

NAME

Create - start terminal emulation

SYNOPSIS

```
#include <courier/Gap3_de.h>
#include <courier/except.h>

void
Create(_Connection, _BDTprocptr, sessionParameterHandle, transportList, createTimeout, credentials,
verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    SessionParameterObject sessionParameterHandle;
    T_p3_3_18 transportList;
    Cardinal createTimeout;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **Create()** function is used to initiate a terminal emulation session with a mainframe computer system. Terminal devices such as TTY, VT100, IBM 3270, etc. can be emulated. This makes it possible for nonXNS terminal devices to interconnect with an XNS system and access XNS services. For this reason, the Gateway Access Protocol (GAP) is also referred to as the Virtual Terminal Protocol (VTP).

When a workstation client requests an emulation session with a host computer, the workstation uses the Clearinghouse to locate an External Communication Service (ECS) that supports connections with a specified host. The workstation then connects to the ECS which initiates a session with the remote host and performs conversions between the XNS and remote host protocols. The protocol conversion provided by the ECS allows information originating from the mainframe computer or anywhere else on the XNS internet to be transferred to and from the mainframe environment and the system on which **Create()** was invoked.

The **sessionParameterHandle** argument is a structure that supplies the host system all the pertinent information necessary for the local workstation to emulate a specific terminal. Acceptable terminal types are: **XEROX800, XEROX850, XEROX860, SYSTEM6, CMCLL, IBM2770, IBM2770HOST, IBM6670, IBM6670HOST, IBM3270, IBM3270HOST, OLDTTYHOST, OLDTTY, OTHERSESSIONTYPE, UNKNOWN, IBM2780, IBM2780HOST, IBM3780, IBM3780HOST,SIEMENS9750, SIEMENS9750HOST, TTYHOST, and TTY.**

Some of these terminal types require some additional information. **XEROX860** requires pollProc. **IBM6670HOST** requires the block size of the transmit and receive packets. **OLDTTY** requires the length of a byte (five, six, seven, or eight bits to a byte), parity (none, odd, even, one, or zero), the stop bit (oneStopBit, twoStopBits), and the frameTimeout (integer indicating milliseconds). **IBM3780HOST** requires the block size of the transmit and receive packets. And **TTY** has the same requirements as **OLDTTY**, plus flowControl (flowControlNone or XOn/XOff).

The **transportList** argument specifies the device that is to receive data. Devices include a modem on an RS232 line, teletype, various BSC terminals and controllers, and so on. Acceptable types are: **RS232C, BSC, TELETYPE, POLLEDBSCCONTROLLER, POLLEDBSCTERMINAL, SDLCCONTROLLER, SDLCTERMINAL, SERVICE, UNUSED, POLLEDBSCPRINTER**, and **SDLCPRINTER**.

The **createTimeout** argument is an integer that specifies the number of seconds to wait for a connection to the mainframe computer before aborting the attempt.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The credentials may be either simple or strong. The client cannot switch from simple to strong authentication or visa versa within the same session. The **verifier** argument is the simple verifier acquired at the same time as the credentials .

RETURN VALUE

This function returns **void**.

ERRORS

Reports [badAddressFormat, controllerAlreadyExists, controllerDoesNotExist, dialingHardwareProblem, illegalTransport, inconsistentParams, mediumConnectFailed, noCommunicationHardware, noDialingHardware, terminalAddressInUse, terminalAddressInvalid, tooManyGateStreams, transmissionMediumUnavailable, serviceTooBusy, userNotAuthenticated, userNotAuthorized, serviceNotFound]

Inbasket2_ChangeBodyPartsStatus

NAME

ChangeBodyPartsStatus - update the status of message body parts

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

ChangeBodyPartsStatusResults
ChangeBodyPartsStatus(_Connection, _BDTprocptr, index, setStatusTo, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    LongCardinal index;
    BodyPartsStatusChangeSequence setStatusTo;
    Session session;
```

DESCRIPTION

The **ChangeBodyPartsStatus()** function is used to update the status of one or more message body parts. When all the body parts of a message have been set to **deletable**, the entire message will be deleted by the mail service. Therefore, if the client wants data from a body part, be sure to store the data before the status is changed to **deletable**. Once the status of a part part has been changed to deletable, it is irreversible. All the parts of a message are accessible until the entire message is deleted. This function also updates the **MessageStatus** field to **KNOWN**.

The **index** argument is the index number of the message to be updated. The **setStatusTo** argument is a structure having two members: **bodyPartIndex** and **deletable**. Together, they define the body parts to be modified. The **bodyPartIndex** member specifies the part in accordance to the MailTransportEnvelopeFormat. The **deletable** member is a an enumerated type that may contain one of two values: **TRUE** or **noChange**.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns a structure called **ChangeBodyPartsStatusResults**. Its one member, **deleted**, is a Boolean that indicates the success of the operation. A value of **TRUE** indicates that all the body parts of the message have been marked as being deletable.

ERRORS

Reports [AuthenticationError[problem], IndexError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

Inbasket2__ChangeMessageStatus

NAME

ChangeMessageStatus - change message status from new to known

SYNOPSIS

```
#include <courier/Inbasket2__de.h>
#include <courier/except.h>

void
ChangeMessageStatus(__Connection, __BDTprocptr, range, changeUserDefinedStatus,
newUserDefinedStatus, session)
    COURIERFD __Connection;
    int (* __BDTprocptr)();
    Range range;
    Boolean changeUserDefinedStatus;
    Cardinal newUserDefinedStatus;
    Session session;
```

DESCRIPTION

The **ChangeMessageStatus()** function is used to update a specified range of messages from new to known. This function may also be used to update the userDefinedStatus.

The **range** argument specifies a set of messages whose status is to be updated from new to known. It requires two parameters. The two parameters are integers that specify the low and high message indices between which all the messages are to be changed. The messages corresponding to the low and high indices will also be affected. The constant **nullIndex** may be used as a value for one or both of the parameters. For example, if the value of **range** is (**nullIndex**, 5) then all the messages between the first inbasket message up to the fifth, inclusive, are updated to known. If (5, **nullIndex**) is specified, then all the messages between the fifth and last, inclusive, are affected. A value of (**nullIndex**, **nullIndex**) may be specified to indicate that all the messages in the inbasket are to be affected. Once a message has been updated to known, it can never be reverted back to new. Attempts to do will be ignored.

The **changeUserDefinedStatus** argument is a Boolean value that indicates whether or not the user defined status should be changed. When set to **TRUE**, **changeUserDefinedStatus** causes **existenceOfMessage** to be set to **KNOWN** and **userDefinedStatus** to be updated with the value of **newUserDefinedStatus**. The default is **FALSE**.

The **newUserDefinedStatus** argument is an integer that specifies the new value of the messages specified in the **range** argument. This user defined status is not interpreted by the mail service. It serves only for use by sophisticated clients to attach arbitrary status information to a message. Only the client who attaches the status information may retrieve it. The range of acceptable values are cardinal numbers between 0 and 65,535, inclusive.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], IndexError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

Inbasket2_Delete

NAME

Delete - remove messages from the inbasket

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

void
Delete(_Connection, _BDTprocptr, range, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Range range;
    Session session;
```

DESCRIPTION

The **Delete()** function is used to remove one or more contiguous messages from the inbasket. If there are no messages within the range of specified indices, no error is returned.

The **range** argument specifies the set of messages to be deleted. It requires two parameters. The two parameters are integers that specify the low and high message indices between which all the messages are to be deleted. The messages corresponding to the low and high indices will also be deleted. The constant **nullIndex** may be used as a value for one or both of the parameters. For example, if the value of range is (**nullIndex**, 5) then all the messages between the first inbasket message up to the fifth, inclusive, will be deleted. If (5, **nullIndex**) is specified, then all the messages between the fifth and last, inclusive, will be deleted. A value of (**nullIndex**, **nullIndex**) may be specified to indicate that all the messages in the inbasket are to be deleted.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], Courier Errors]

Inbasket2_GetSize

NAME

GetSize - retrieve the size of the inbasket

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

GetSizeResults
GetSize(_Connection, _BDTprocptr, inbasket, credentials, verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName inbasket;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **GetSize()** function is used to retrieve a tally of the disk space occupied by all the messages in an inbasket The value returned is in units of bytes.

The **inbasket** argument is a structure of type **ThreePartName**. Its three members, **organization**, **domain**, and **object**, identify the mail recipient. The recipient must be registered with the Clearinghouse. Usually the value of the **inbasket** argument is the same as the user identified in the credentials. UNIX wildcards may not be used to specify any part of the name. Aliases are allowed and are resolved by the Mail Service.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The **verifier** argument is the verifier returned earlier by the Authentication Service.

RETURN VALUE

This function returns a structure called **GetSizeResults**. Its one member, **sizeInBytes**, is a cardinal number that indicates the total number of bytes being used by the specified inbasket.

ERRORS

Reports [AuthenticationError[problem], AccessError[problem], OtherError[problem], ServiceError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

Inbasket2_Logon, _Logoff

NAME

Logon - initiate a new inbasket session
Logoff - terminate an inbasket session

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

LogonResults
Logon(_Connection, _BDTprocptr, inbasket, credentials, verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName inbasket;
    Credentials credentials;
    Verifier verifier;

void
Logoff(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **Logon()** function is used to initiate a new inbasket session with the mail service. Once an inbasket session has been initiated, the client may access messages sent to the user specified in the credentials. The **Logoff()** function is used to end an inbasket session with the mail service. The inbasket session handle will then become invalid.

Most inbasket operations take place within the context of a session. Each session references a single inbasket that is specified when the session is initiated. The name of the inbasket will be the same as the name of the message recipient. The message recipient does not have to be same person as specified in the credentials that were used to authorize the inbasket session. More than one session may access the same inbasket simultaneously. When this occurs, each session is cognizant of changes made by the other session(s).

The **inbasket** argument is a structure comprised of **organization**, **domain**, and **object** name. It is used to identify the mail recipient for whom an inbasket session is being initiated. The recipient must be registered with the Clearinghouse. Usually the value of the **inbasket** argument is the same as the user identified in the credentials. UNIX wildcards may not be used to specify any part of the name. Aliases are allowed and are resolved by the Mail Service. The name is to be the same as the **inbasket** name that was initially assigned by the System Administrator.

The **credentials** argument is the credentials returned earlier by the Authentication Service. This argument is used to by the Authentication Service to unequivocally determine a client's right to access the specified inbasket. The credentials may be either simple or strong credentials. If the user specified in the **inbasket** argument is not the same as that identified by the credentials, the client must have strong credentials to initiate the inbasket session. The **verifier** argument is the verifier returned by the Authentication Service.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

The **Logon()** function returns a structure called **LogonResults**. It contains three members: **session**, **state**, and **anchor**. **session** is of type **Session**. Its one member, **token**, is of type **T_r18_2_63**. token is an unspecified array that is used as an identifier. It should be passed unchanged to all operations within the same session.

state is a structure of type **State**. It contains two members, **new** and **total**. **new** is a cardinal number that indicates the number of new, or unread, messages in the inbasket. **total** is a cardinal number that indicates the sum total of all the messages in the inbasket. The values returned by new and total reflect only those changes that have been made permanent.

anchor is a five-wide integer of type **Anchor**. It is used to determine the validity of the mailing service cached indices. Each message in an inbasket is identified by a unique index which is permanently assigned to each message. Indices are positive integers allocated from a 32-bit field. On occasion, the association of an index to a message becomes invalid due to such events as, shuttling an inbasket between mail services.

The **anchor** is especially important when the same message indices are used in more than one inbasket session. If the same indices are used, the **anchor** returned by each call to **Logon()** should be stored and compared against the anchors returned by each succeeding call to **Logon()**. If the **anchor** returned by a call to **Logon()** is different than that of a preceding call to **Logon()**, then the previously cached indices are invalid. When this occurs, flush the old values from the cache(s) in order to maintain accurate indices of the messages in the mail box. When a client is aware that an index has become invalid during the course of an inbasket session, the client may assume that the message referenced by the invalid index has been deleted by another client.

The **Logoff()** function returns **void**.

ERRORS

Logon() reports [AccessError[problem], AuthenticationError[problem], InbasketInUse, OtherError[problem], ServiceError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]. **Logoff()** reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

Inbasket2_MailCheck

NAME

MailCheck - check an inbasket from within a session

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

MailCheckResults
MailCheck(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **MailCheck()** function is used to determine the state of an inbasket. Unlike **MailPoll()**, this function is to be used during an inbasket session.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns a structure called **MailCheckResults**. Its one member, **state**, is of type **State**. It is a structure having two members: **new** and **total**. **new** is a cardinal number that indicates the number of new, or unread, messages in the inbasket. **total** is a cardinal number that indicates the sum total of all the messages in the inbasket. The values returned by **new** and **total** reflect only those changes that have been made permanent.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

SEE ALSO

Logon()

Inbasket2_MailPoll

NAME

MailPoll - check an inbasket without starting a session

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

MailPollResults
MailPoll(_Connection, _BDTprocptr, inbasket, credentials, verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName inbasket;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **MailPoll()** function is used to quickly determine the state of an inbasket. This function is fast because the client does not incur the overhead of initiating an inbasket session.

The **inbasket** argument is a structure of type **ThreePartName**. Its three members, **organization**, **domain**, and **object**, identify the mail recipient. The recipient must be registered with the Clearinghouse. Usually the value of the **inbasket** argument is same as the user identified in the credentials. UNIX wildcards may not be used to specify any part of the name. If the object name encountered is an alias, it is de-referenced before it is processed. The Mail Service will resolve aliases.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The **verifier** argument is the verifier returned earlier by the Authentication Service.

RETURN VALUE

This function returns a structure called **MailPollResults**. Its one member, **state**, is a structure of type **State**. **state** has two members: **new** and **total**. **new** is a cardinal number that indicates the number of new, or unread, messages in the inbasket. **total** is a cardinal number that indicates the sum total of all the messages in the inbasket. The values returned by **new** and **total** reflect only those changes that have been made permanent.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], OtherError[problem], SessionError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

SEE ALSO

MailCheck()

Inbasket2__RetrieveBodyParts

NAME

RetrieveBodyParts - extract specific body parts of a message

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

void
RetrieveBodyParts(_Connection, _BDTprocptr, message, bodyParts, contents, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    LongCardinal message;
    BodyPartSequence bodyParts;
    BulkData1_Descriptor contents;
    Session session;
```

DESCRIPTION

The **RetrieveBodyParts()** function is used to copy specific parts of an inbasket message from the outbasket to the client's local disk.

The **_BDTprocptr** argument is the name of the client-defined function that will be retrieving the body parts. This function must be provided.

The **message** argument is the index number of the message from which body parts are to be retrieved. The **bodyParts** argument may be either a list of individual body parts or the constant **allBodyParts**. Body parts are retrieved in the order they are specified in the **BodyPartSequence** parameter.

The **contents** argument is the stream of body part data in which the returned parts are to be placed in accordance to the Bulk Data Transfer Protocol. The returned body parts are concatenated without any structure-related information separating them. The client can determine the starting point of each body part by using the part sizes listed in the tableOfContents. Use **BulkData1_immediateSink** when retrieving to a local file.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], IndexError[problem], OtherError[problem],
SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors:
REJECT__ERROR, SYSTEM__ERROR, default]

Inbasket2_RetrieveEnvelopes

NAME

RetrieveEnvelopes - extract the envelope of a message

SYNOPSIS

```
#include <courier/Inbasket2_de.h>
#include <courier/except.h>

void
RetrieveEnvelopes(_Connection, _BDTprocptr, index, whichMsg, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    LongCardinal index;
    T_p18_2_69 whichMsg;
    Session session;
```

DESCRIPTION

The **RetrieveEnvelopes()** function is used to extract a particular message (envelope) from the inbasket. For messages of type **StandardMessage**, the heading (body part number 0) is also extracted. One message, a series of messages, or all the messages in the inbasket may be easily retrieved.

The **index** argument is the index number of the message to be enumerated. The constant **nullIndex** may be used to enumerate all the messages in the inbasket. To enable **nullIndex**, **whichMsg** must be set to **next**. **nullIndex** is also returned when there are no more messages to enumerate. If there is no message in the inbasket that has the specified **index** value, IndexError is raised.

The **whichMsg** argument is an enumerated type that determines which of two messages to enumerate. One of two values may be specified, **this** or **next**. **this** indicates that the message having the number specified in the **index** argument is to be enumerated. **next** indicates that the message after the message having the number specified in the **index** argument is to be enumerated.

To view a series of messages, specify the **index** number of a message that immediately precedes the messages of interest and set **whichMsg** to **next**. After the first call to **RetrieveEnvelopes()**, set the value of **index** in the current call to the value of **index** returned by the previous call. This will cause all the messages starting from **index** + 1 and ending with the last message to be enumerated.

The **session** argument is the inbasket session handle returned by a preceding call to **Logon()**.

RETURN VALUE

This function returns a structure called **RetrieveEnvelopesResults**. It has three members: **transportEnvelope, status** and **index**.

transportEnvelope is of type **Envelope**.It is an array of records that defines the MTA-visible portions of the message. It contains information regarding the pre-delivery history of the message.

status is of type **Status**. It is a structure that has two members: **MessageStatus** and **BodyPartsStatus**.

MessageStatus is a structure that applies to the message as a whole. It indicates one of two conditions: **NEW** or **KNOWN**. **NEW** means the client has not yet been made aware of the delivery of the specific message. **KNOWN** means the client has been appraised of the delivery of the specific message but has not yet received the message contents. It is important to note that status serves as an indication of the reception of a message by the client, as mediated by the mail service. It does not reflect the delivery of the message itself to the client by the message transfer service, nor the transfer of information to the inbasket. After being alerted as to the status of a message, it is the client's responsibility to to update **MessageStatus**.

BodyPartsStatus refers to, and indicates the condition of the sequence of component parts that comprise an entire message. **BodyPartsStatus** is an array of Boolean values, with one Boolean value per each message body part. The body parts of a message are numbered from zero to the actual number of body parts minus one. Body parts are numbered in the same order as they are displayed in **tableOfContents**. **BodyPartIndex** is used to refer to a specific part of a message.

index is a cardinal number that indicates the index number of the last enumerated message. When the value of **index** is **nullIndex**, it indicates that there are no more messages to enumerate.

ERRORS

Reports [AuthenticationError[problem], IndexError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

MailTransport5__AbortRetrieval

NAME

AbortRetrieval - postpone delivery of a message

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

void
AbortRetrieval(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **AbortRetrieval()** function is used to direct the mail service to stop the retrieval process immediately and retain the remainder of the message until the client is ready to accept it. Subsequent messages will become unavailable until the envelope or message in question is disposed of in some way. This function may only be called immediately following a call to either **RetrieveEnvelope()** or **RetrieveContent()**.

The **session** argument is the session handle returned by a preceding call to **BeginRetrieval()**.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

MailTransport5_BeginPost

NAME

BeginPost - post outgoing messages

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

BeginPostResults
BeginPost(_Connection, _BDTprocptr, postingData, postIfInvalidNames, allowDLRecipients,
optionalEnvelopeData, credentials, verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    PostingData postingData;
    Boolean postIfInvalidNames;
    Boolean allowDLRecipients;
    T_p17_5_38 optionalEnvelopeData;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **BeginPost()** function is used to initiate the posting of a message with a mail service. Once an appropriate mail service has been located, **BeginPost()** is used to start the posting of a message. It is then followed by a sequence of **PostOneBodyPart()** operations, and terminated by **EndPost()**.

The **postingData** argument is a structure that contains a list of clients to receive the message, the contents type, the size of the message contents, and a list of the body parts comprising the message. **postingData** contains four members: **Recipients, contentsType, contentsSize,** and **bodyPartTypesSequence**.

> **Recipients** is a sequence of records, each one defining the full path name of each intended recipient.
>
> **contentsType** is an enumerated type that directs the interpretation of messages by the Mail Service. One of the following values may be specified:
>
> > **UNSPECIFIED**
> > **STANDARDMESSAGE**
> > **REPORT**
> > **NULL**
> >
> > To be in a human readable format, the value of **contentsType** must be either **REPORT** or **STANDARDMESSAGE**.
>
> **contentsSize** specifies the size of the entire contents portion of the message in bytes. If the size specified is not within 5000 bytes of actual size, the Mail Service will raise an error.
>
> **bodyPartTypesSequence** is a sequence of body part types. Body part types for standard messages are a sequence of cardinal numbers that range between 0 and 12, inclusive. 0 indicates the Heading body part. Number 1-12 indicate portions of the Interpersonal Message body part. There should be a one-to-one correspondence between the body parts in **T_r17_5_37** and the elements in the message.

The **postIfInvalidNames** argument is a Boolean value that controls how **BeginPost()** handles invalid recipients. A value of **TRUE** causes all valid recipients to receive the message, regardless of the number of non-valid recipients. The invalid recipient names will be returned. A value of **FALSE** will prevent the message from being sent to anyone if an invalid recipient name exists and an results in an error.

The **allowDLRecipients** argument is a Boolean that indicates whether or not the message is to be sent to distribution lists. A value of **TRUE** causes the message to be sent regardless of the number of intended recipients. A value of **FALSE** causes any recipient that is a distribution list to be designated as invalid.

The **optionalEnvelopeData** argument is a structure that allows the client to include additional information regarding the handling of the message.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The **verifier** argument is the verifier returned earlier by the Authentication Service.

RETURN VALUE

This function returns a structure called **BeginPostResults**. It has two members: **session** and **invalidNames**.

session is of type **Session**. It is a mail transport session handle to be used to complete the posting process.

invalidNames is of type **InvalidNameList**. It is a list of invalid recipients, in the case where not all the recipients are valid but the message was posted anyway. This can only occur when **postIfInvalidNames** is set to **TRUE**. The mail transport session handle returned will be invalid if **postIfInvalidNames** was set to **FALSE** and the recipient list contained invalid names. An error will be raised if an invalid mail transport session handle is passed to **PostOneBodyPart()**.

ERRORS

Reports [AuthenticationError[problem], InvalidRecipients[namelist], OtherError[problem], ServiceError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

MailTransport5__BeginRetrieval

NAME

BeginRetrieval - initiate the extraction of messages from a mail slot

SYNOPSIS

```
#include <courier/ MailTran5_de.h>
#include <courier/except.h>

BeginRetrievalResults
BeginRetrieval(_Connection, _BDTprocptr, deliverySlot, credentials, verifier)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    ThreePartName deliverySlot;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **BeginRetrieval()** function is used to initiate the retrieval of one or more messages from a specified delivery slot. A typical delivery slot session consists of a call to **BeginRetrieval()**, multiple calls to **RetrieveEnvelope()** and **RetrieveContent()**, and a concluding call to **EndRetrieval()**.

The **deliverySlot** argument identifies the slot to be accessed. A delivery slot is associated with a specific recipient name. To retrieve messages via a delivery slot, the client must locate the mail service on which the recipient name is registered. This is done by looking up the recipient's name in the Clearinghouse to determine the value of the mailboxes property associated with that name. (The format of this property is defined in the Clearinghouse Entry Formats Standard.) The mailboxes property contains an array of mail service names, each of which hold a mailbox for the user.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The credentials may be either simple or strong. If the delivery slot is not owned by the client identified in the credentials, the client must have strong credentials to access that slot. The client cannot switch from simple to strong authentication or visa versa within the same session.

The **verifier** argument is the simple verifier acquired at the same time as the credentials .

RETURN VALUE

This function returns a structure called **BeginRetrievalResults**. Its one **member**, session, is of type **Session**. It is the mail transport session handle that is to be passed to all related ***Retrieval()** functions.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], OtherError[problem], ServiceError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

MailTransport5_EndPost

NAME

EndPost - signal termination of posting a message

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

EndPostResults
EndPost( _Connection, _BDTprocptr, session, abortPost)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
    Boolean abortPost;
```

DESCRIPTION

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

The **session** argument is the transport session handle returned by **BeginPost()**. Once the call to **EndPost()** completes, the mail transport session handle is no longer valid. The **abortPost** argument is a Boolean that indicates what is to be done with the completed message. If **abortPost** is **TRUE**, the message will not be posted and it will be discarded by the mail service. If set to **FALSE**, the message will be sent to the recipients listed in **BeginPost()**. The default is **FALSE**.

RETURN VALUE

This function returns a structure of type **EndPostResults**. Its one member, **messageID**, is of type **MessageID**. **messageID** is a unique identifier assigned by the mail service during posting and is used for use in pairing messages to their associated reports or in locating messages referenced by other messages.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Error]

MailTransport5_EndRetrieval

NAME

EndRetrieval - terminate a retrieval sequence

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

void
EndRetrieval(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **EndRetrieval()** function is used to end the current delivery slot retrieval sequence. Calling this function invalidates the session handle returned by a preceding call to **BeginRetrieval()**.

The **session** argument is the session handle returned by a preceding call to **BeginRetrieval**().

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

MailTransport5_MailPoll

NAME

MailPoll - check a delivery slot for messages

SYNOPSIS

```
#include <courier/ MailTran5_de.h>
#include <courier/except.h>

MailPollResults
MailPoll ( _Connection, _BDTprocptr, deliverySlot, credentials, verifier)
    COURIERFD _Connection;
    int (* _BDTprocptr)();
    ThreePartName deliverySlot;
    Credentials credentials;
    Verifier verifier;
```

DESCRIPTION

The **MailPoll()** function is used to determine if messages are present in a delivery mail slot. Due to the overhead of invoking the various ***Retrieval()** functions, **MailPoll()** is the suggested means of verifying the existence of mail.

The **deliverySlot** argument identifies the slot to be accessed. A delivery slot is associated with a specific recipient name. To retrieve messages via a delivery slot, the client must locate the mail service on which the recipient name is registered. This is done by looking up the recipient's name in the Clearinghouse to determine the value of the mailboxes property associated with that name. (The format of this property is defined in the Clearinghouse Entry Formats Standard.) The mailboxes property contains an array of mail service names, each of which may contain mail for the specified recipient.

The **credentials** argument is the credentials returned earlier by the Authentication Service. The credentials may be either simple or strong. If the delivery slot is not owned by the client identified in the credentials, the client must have strong credentials to access that slot. The client cannot switch from simple to strong authentication or visa versa within the same session.

The **verifier** argument is the simple verifier acquired at the same time as the credentials .

RETURN VALUE

This function returns a structure called **MailPollResults**. Its one member, **mailPresent**, is a Boolean value that indicates the presence of mail in the delivery slot. A value of **TRUE** indicates there is mail ready for retrieval. A value of **FALSE** indicates that the delivery slot is empty.

ERRORS

Reports [AccessError[problem], AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

MailTransport5_PostOneBodyPart

NAME

PostOneBodyPart - send message data to a mail service

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

void
PostOneBodyPart(_Connection, _BDTprocptr, session, bodyPartType, contents)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
    LongCardinal bodyPartType;
    BulkData1_Descriptor contents;
```

DESCRIPTION

The **PostOneBodyPart()** function is used to submit the data that was declared in **BeginPost()** to a mail service. **BeginPost()** informs the mail service that a specific body of data, having specific qualities, is to follow. **PostOneBodyPart()** specifies that body of data. This function is to be called once for each body part. If more than five minutes elapses between the time **BeginPost()** is called and **PostOneBodyPart()** is called, an error is raised.

The **_BDTprocptr** argument is the name of the client-defined function that will be posting the body part. This function must be provided.

The **session** argument is the transport session handle returned by **BeginPost()**. The **bodyPartType** argument is a cardinal number that indicates the body part. This argument insures that all the body parts that are supposed to be included in the message are sent to the mail service. The value of this argument is to be identical with the value of **bodyPartTypeSequence** that was specified in **BeginPost()**. If there is any discrepancy between **BeginPost()**, **bodyPartTypeSequence**, and **bodyPartType**, an error is raised.

The **contents** argument specifies the source that is to supply the data comprising the message in accordance to the Bulk Data Transfer Protocol.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

MailTransport5_RetrieveContent

NAME

RetrieveContent - extract the contents of a message

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

void
RetrieveContent(_Connection, _BDTprocptr, content, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    BulkData1_Descriptor content;
    Session session;
```

DESCRIPTION

The **RetrieveContent()** function is used to extract the contents of a message that is at the head of the delivery slot queue. This function must follow a successful call to **RetrieveEnvelope()**.

The **_BDTprocptr** argument is the name of the client-defined function that will retrieve the contents of the message. This function must be provided.

The **content** argument is the sink that is to receive the contents in accordance to the Bulk Data Transfer Protocol. Use **BulkData1_immediateSink** when retrieving to a local file.

The **session** argument is the session handle returned by a preceding call to **BeginRetrieval()**.

RETURN VALUE

This function returns **void**.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

MailTransport5__RetrieveEnvelope

NAME

RetrieveEnvelope - extract the header information regarding a delivery slot message

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

RetrieveEnvelopeResults
RetrieveEnvelope(_Connection, _BDTprocptr, session)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    Session session;
```

DESCRIPTION

The **RetrieveEnvelope()** function is used to extract an envelope from the head of the delivery slot queue and, for those messages with contents of type ctStandardMessage, the heading (body part #0) is also extracted. The extracted envelope does not contain the message itself, only pertinent information regarding the message. If, based upon the envelope information, the message is of no interest, the envelope contents may be discarded by immediately calling **RetrieveEnvelope()** a second time.

This function may be called repeatedly in tandem with **RetrieveContent()** during the same session to extract all the envelopes in a delivery slot.

The **session** argument is the session handle returned by a previous call to **BeginRetrieval()**.

RETURN VALUE

This function returns a structure called **RetrieveEnvelopeResults**. It has two members: **empty** and **envelope**.

empty is a Boolean value that indicates the presence of available envelopes. A value of **TRUE** indicates that the active delivery slot is empty and there are no envelopes. A value of **FALSE** indicates that the delivery slot is not empty and there are envelopes available.

envelope is of type **Envelope.** It is the envelope itself.

ERRORS

Reports [AuthenticationError[problem], OtherError[problem], SessionError[problem], ServiceError[problem], TransferError[problem], Courier Errors: REJECT__ERROR, SYSTEM__ERROR, default]

MailTransport5_ServerPoll

NAME

ServerPoll - query the mail service if it will accept posted messages

SYNOPSIS

```
#include <courier/MailTran5_de.h>
#include <courier/except.h>

ServerPollResults
ServerPoll(_Connection, _BDTprocptr)
   COURIERFD _Connection;
   int (*_BDTprocptr)();
```

DESCRIPTION

The **ServerPoll()** function is used to determine if the mail service currently in use will accept additional messages for posting. This function may be broadcast to assist other clients in locating a mail service to use for posting messages. This function requires only the **_Connection** and **_BDTprocptr** arguments.

RETURN VALUE

This function returns a structure called **ServerPollResults**. It has three members: **willingness, address,** and **serverName**.

willingness is of type **Willingness**. It is a cardinal number that specifies the availability of a particular mail service for posting mail. The range of **willingness** is between 1 and 10, inclusive. 1 is least receptive to new postings. 10 is most receptive. The returned value for **willingness**[i] gives the service's ability to accept messages of size 8[i] bytes to size (8[i+1] - 1) bytes. The last element in the sequence gives the service's ability to accept a message of size 8(index of the last element in the sequence) bytes to a message of unbounded size,

address is of type **NetworkAddressList**. It is the list of clearinghouse network addresses that may be used to contact a mail service. The **address** list will contain more than one element only if the mail service is connected to more than one network.

serverName is of type **ThreePartName**. It is a full path name that identifies the name of the responding mail service.

ERRORS

Reports [Courier Errors: REJECT_ERROR, SYSTEM_ERROR, default]

Printing3_GetPrinterProperties

NAME

GetPrinterProperties - retrieve the static properties of a printer

SYNOPSIS

```
#include <courier/Printing3_de.h>
#include <courier/except.h>
#include <courier/ papersize.h>

GetPrinterPropertiesResults
GetPrinterProperties(_Connection, _BDTprocptr)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
```

DESCRIPTION

The **GetPrinterProperties()** function is used to query the print service for the static properties of a printer. This function knows the printer to access based upon the value of the **_Connection** argument.

RETURN VALUE

This function returns a structure called **GetPrinterPropertiesResults**. Its one member, **properties**, specifies the handling characteristics of the printer. **properties** is of type **PrinterProperties**. It has three fields: **media, staple**, and **twoSided**.

The **media** field is like the **mediumHint** parameter of PrintOptions in that it specifies the paper sizes available for a specific printer. The media listed need not be immediately available, but the print service should be able to provide them at the time of printing.

The **staple** field specifies if a document sent to the printer in question will be stapled upon completion. The default is **FALSE**.

The **twoSided** argument specifies whether or not a document will be printed on both sides of each sheet of paper. The default is **FALSE**.

ERRORS

Reports [ServiceUnavailable, SystemError, Undefined[problem]]

SEE ALSO

Print()

Printing3_GetPrinterStatus

NAME

GetPrinterStatus - determine the availability of the print service

SYNOPSIS

```
#include <courier/Printing3_de.h>
#include <courier/except.h>
#include <courier/papersize.h>

GetPrinterStatusResults
GetPrinterStatus(_Connection, _BDTprocptr)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
```

DESCRIPTION

The **GetPrinterStatus()** function is used to determine where the print service is with respect to the four aspects of printing a document: spooling, formatting, printing, and the paper on which to print the document. The spooler accepts a master; the formatter decomposes it; and the printer marks the decomposed document for printing. This function knows the printer to access based upon the value of the **_Connection** argument.

RETURN VALUE

The **GetPrinterStatus()** function returns **GetPrinterStatusResults**. Its one member, **status**, describes the state, or condition, of the four aspects of printing. **status** is of type **PrinterStatus**. It contains the status of four objects: spooler, formatter, printer, and media.

The spooler may be in any one of four states: **AVAILABLE**, **BUSY**, **DISABLED**, and **FULL**.

The formatter may be in any one of three states: **AVAILABLE**, **BUSY**, and **DISABLED**.

The printer may be in any one of five states: **AVAILABLE**, **BUSY**, **DISABLED**, **NEEDSATTENTION**, and **NEEDSKEYOPERATOR**.

And **MEDIA** describes the paper sizes that are available on the printer.

AVAILABLE indicates that the respective phase is ready to accept input.

BUSY indicates that the respective phase is currently processing another print request and cannot accept input. This is a transient condition that lasts a relatively short time.

DISABLED indicates that the respective phase is not available and cannot accept input. The duration of this state may last a long time.

FULL indicates that the spooler is currently unable to accept additional input. This is due to the number of printing requests exceeding the capacity of the spooler queue.

NEEDSATTENTION indicates that the marking engine of the printer is not functioning properly and requires some non-technical human intervention.

NEEDSKEYOPERATOR indicates that the marking engine of the printer is not functioning properly and requires the attention of a trained technician.

MEDIA enumerates the paper sizes currently available on the target printer.

ERRORS

Reports [ServiceUnavailable, SystemError, Undefined]

SEE ALSO

Print(), GetPrintRequestStatus(), GetPrinterProperties()

Printing3__GetPrintRequestStatus

NAME

GetPrintRequestStatus - determine the status of an outstanding print request

SYNOPSIS

```
#include <courier/Printing3_de.h>
#include <courier/except.h>
#include <courier/ papersize.h>

GetPrintRequestStatusResults
GetPrintRequestStatus(_Connection, _BDTprocptr, printRequestID)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    RequestID printRequestID;
```

DESCRIPTION

The **GetPrintRequestStatus()** function is used to ascertain the status of a document that was sent to a printer. The **printRequestID** argument is the return value of the **Print()** function and must be supplied here to properly identify the document in question.

RETURN VALUE

This function returns a structure called **GetPrintRequestStatusResults**. Its one member, **status**, is of type **RequestStatus**. It may have one of the following nine values:

PENDING, has not begun printing.
INPROGRESS, currently being printed.
COMPLETED, the document has successfully completed printing.
COMPLETEDWITHWARNING, the document has been printed but low level problems have been encountered.
UNKNOWN, the print service is at a complete loss as to what happened to the document in question.
REJECTED, the document was not accepted into the marking phase because errors, such as illegal interpress commands, have been encountered in the master.
ABORTED, the printing request was aborted during the formatting or marking phase.
CANCELLED, the document was queued for printing but someone having priority credentials cancelled the printing request.
HELD, the printing service has postponed processing the document until a later time because other print requests having a higher priority have been received.

ERRORS

Reports [ServiceUnavailable, SystemError, Undefined[problem]]

SEE ALSO

Print(), GetPrinterProperties(), GetPrinterStatus()

Printing3_Print

NAME

Print - print a master

SYNOPSIS

```
#include <courier/Printing3_de.h>
#include <courier/except.h>
#include <courier/papersize.h>

PrintResults
Print(_Connection, _BDTprocptr, master, printAttributes, printOptions)
    COURIERFD _Connection;
    int (*_BDTprocptr)();
    BulkData1_Descriptor master;
    PrintAttributes printAttributes;
    PrintOptions printOptions;
```

DESCRIPTION

The **Print()** function is used to access bulk transfer data in a source and send it to the print service queue. This function knows the printer to access based upon the value of the **_Connection** argument.

The **_BDTprocptr** argument is the name of the client defined function that will send the data to the print queue. This function must be provided.

The **master** argument is a bulk data transfer parameter that specifies the source that is to supply the interpress master in accordance with the Bulk Data Transfer Protocol. Use **BulkData1_immediateSink** when retrieving to a local file.

The **printAttributes** argument is a structure that specifies supplementary information about the document to be printed, such as the name of the object to be printed, the creation date, and the sender's name. The information specified here is printed on the document cover page.

The **printOptions** argument specifies parameters that affect the printing of a document, and the characteristics the printed document is to have or that are relevant to the printing process. There are ten parameters to this argument:

> The first, **printObjectSize**, indicates the size of the master to be printed in bytes. This may be useful for allocating printing resources or determining if the master size exceeds the capability of the printer. The default value is the size of the master received.
>
> The second parameter, **recipientName**, is the name of the person for whom the printed document is intended. Typically this will appear on the banner sheet of the printed document and, on print servers with sorters, it may be used as the basis of a sort key. The default value is the same as the senderName parameter specified in the **printAttributes** argument.
>
> The **message** parameter is typically used to specify the status information to be displayed either locally or printed on the banner sheet. This message accompanies a print request. It is a text string. The default is a **NULL** string.
>
> The **copyCount** parameter specifies the number of copies to be made. The default is 1.
>
> The **pagesToPrint** parameter specifies the range of pages to be printed. **beginningPageNumber** specifies the page number at which printing is to commence.

The **endingPageNumber** parameter specifies the page number at which printing is to stop. "Page number" is the ordinal position of the page in the document, not the page number actually printed on the page. The default value is every page in the master.

The **mediumHint** parameter indicates the size of the paper on which the document is to be printed. Refer to papersize.h and Printing3.cr for information regarding acceptable paper sizes and the format for specifying them. Though listed as an option, the value of unknown may not be used. The default paper size is specific to each print service.

The **priorityHint** parameter is the printing priority requested by the sender. The default value is specific to each print service. If a request is not to be processed immediately by the print service a non-**NULL** releaseKey is to be supplied by the user.

The **releaseKey** parameter is datum that must be presented to the print service in order to release a held request. The source for a releaseKey is assumed to be a password consisting of a string of characters. The **releaseKey** is computed from the password using the algorithm specified by the Authentication Protocol. The default value is 177777B, "**NULL** string".

The **staple** parameter specifies if the document is to be stapled upon completion. The default is **FALSE**.

The **twoSided** parameter specifies whether or not the document is to be printed on both sides of each sheet of paper. The default is **FALSE**.

RETURN VALUE

This function returns **PrintResults**, a structure whose one member, **printRequestID**, contains a unique identifier for the document being printed. The identifier is assigned by the print service and is of type **RequestID**. It may later be supplied as an argument to **GetPrintRequestStatus()**, a function that ascertains the status of documents that have been sent to the print spooler.

ERRORS

Reports [Busy, ConnectionError, InsufficientSpoolSpace, InvalidPrintParameters, MasterTooLarge, MediumUnavailable, ServiceUnavailable, SpoolingDisabled, SpoolingQueueFull, SystemError, TooManyClients, TransferError[problem], Undefined[problem]]

SEE ALSO

GetPrintRequestStatus(), GetPrinterProperties(), GetPrinterStatus()

Index

NOTES

NOTES

NOTES

NOTES

Customer Comments

XEROX®

VP Series Reference Library
Document Interfaces Toolkit Software Reference

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

Printed in U.S.A.