Display PostScript System

Adobe Systems Incorporated

Display PostScript Toolkit for X

15 April 1993

Adobe Systems Incorporated

Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/

Contents

List of Figures

List of Tables

List of Examples

Display PostScript Toolkit for X

1 About This Manual

Display PostScript Toolkit for X manual describes the Display PostScript Toolkit for the X Window System. It also contains information about locating PostScript language resources and about the *makepsres* utility.

The Display PostScript Toolkit is a collection of procedures and objects for programmers who use the Display PostScript extension to the X Window System, which is sometimes referred to as DPS/X. The toolkit can be used for context management, user object management, user path handling, and file previewing. It also allows you to preview and choose from currently available fonts by using the font selection panel and the font sampler.

The toolkit is supplemented by procedures for locating PostScript language resources using resource database files and by the *makepsres* utility, which can be used to create the resource database files. These utilities are used by the font selection panel, but can be helpful in other situations as well.

The toolkit library that contains the facilities described in this manual is available from several sources:

- The X Consortium Release 5 contributed software under *contrib/lib/DPS*.
- The Display PostScript System Software Development Kit for the X Window System, available from Adobe Systems.
- On Adobe's public access file server. Using the file server is described in the preface of this book.
- The Display PostScript system release provided by your system vendor. Note, however, that not all system vendors include the Display PostScript Toolkit as part of their release.

TK

1.1 What This Manual Contains

Section 2, "About the Display PostScript Toolkit," introduces the Display PostScript Toolkit.

Section 3, “Context Management Procedures,” describes context management procedures.

Section 4, “User Objects,” documents facilities for working with user objects.

Section 5, “User Paths,” introduces a convenient interface for working with user paths.

Section 6, “File Preview Procedures,” describes file preview procedures, which simplify rendering PostScript language files into X drawable objects (windows or pixmaps).

Section 7, “The Motif Font Selection Panel,” provides information about the font selection panel, which can be used to view and select the fonts available on a workstation.

Section 8, “The Motif Font Sampler,” provides information about the font sampler, which can be popped up from the font selection panel for viewing multiple fonts simultaneously.

Appendix AA explains how applications can locate PostScript language resources using resource database files.

Appendix BB documents the *makepsres* utility, which can be used to create resource database files.

2 About the Display PostScript Toolkit

The toolkit is located in the libraries *libdpstk.a* and *libdpstkXm.a*. The *libdpstkXm.a* library contains the Motif font selection and font sampler dialogs, and the library *libdpstk.a* contains everything else. When compiling an application, you must specify these libraries to the linker before the Display PostScript library *libdps.a*.

- If an application uses the font selection panel, you must specify the toolkit libraries to the linker before the Motif™ library. In that case, you must also link with the resource location library for the PostScript language, which is described in Appendix A. The normal order for libraries is:

```
-ldpstk -ldpstkXm -lpsres -lXm -lXt -ldps -lX11 -lm
```

- If the application does not use the font selection panel, linking with *libdpstkXm.a* and Motif is not required. In that case, the normal order for libraries is:

```
-ldpstk -ldps -lX11 -lm
```

Note: The math library, libm.a, is required by some implementations of libdps.a.

2.1 Common Definitions

The header file *<DPS/dpsXcommon.h>* contains definitions used by various procedures in the Display PostScript Toolkit.

2.1.1 Type Definitions

The type *DPSPointer* is used for pointers of an unspecified type.

DPSPointer

```
typedef char *DPSPointer;
```

Note: The definition of DPSPointer is implementation-specific.

TK

2.1.2 Return Values

Table 1 describes the values returned by the procedures in the Display PostScript Toolkit. These values are all of type *int*.

Table 1 *Toolkit return values*

Return Value	*Meaning*
dps_status_success	The procedure executed successfully and to completion.
dps_status_failure	The procedure failed. The reason is documented in the description of the procedure.
dps_status_no_extension	The procedure attempted to execute an operation that requires context creation and discovered that the server does not support the Display PostScript extension.
dps_status_unregistered_context	The procedure requires a context registered with the context manager, and the passed context has not been registered.
dps_status_illegal_value	One of the parameters to the procedure has an illegal value.
dps_status_postscript_error	The PostScript language code being handled by the procedure contains an error.
dps_status_imaging_incomplete	The PostScript language code being handled by the procedure did not finish execution within a time-out period.

3 Context Management Procedures

In DPS/X, a context is a server resource that represents all of the execution state needed by the PostScript interpreter to run PostScript language programs. Contexts are described in *Client Library Reference Manual* and in *Client Library Supplement for X*.

This section documents context management procedures provided by the Display PostScript Toolkit. A brief introduction is followed by a table listing all available procedures. The rest of the section lists structures and procedure definitions in alphabetical order.

3.1 Introduction

A PostScript execution context consists of all the information (or state) needed by the PostScript interpreter to execute a PostScript language program. Context management utilities allow different code modules to share PostScript contexts. They make it easy to associate several drawables with one context and to switch between the drawables. They also hide the details of context creation from an application by creating and managing default contexts.

Some libraries provide an encapsulated service—a closed, well-defined task with minimal outside interaction (for example, displaying read-only text). If you are writing a library that provides an encapsulated service, the context management procedures can simplify the interface you provide to applications. For example, the file preview procedures described in section 6, “File Preview Procedures,” can use the context management procedures to get a context for previewing a file. The font selection panel described in section 7, “The Motif Font Selection Panel,” can use the context management procedures to get a context for previewing fonts. If an application uses both file previewing and the font selection panel, they can share the same context. The shared context is called the *default context* for the application. Context management procedures allow an application that uses the file preview procedures to ignore contexts completely; the application does not even have to know that contexts exist.

Code that uses the context management procedures must include *<DPS/dpsXshare.h>*, which automatically includes *<DPS/dpsXcommon.h>*.

An application can get the shared context for a display by calling **XDPSGetSharedContext**. To use the context management procedures on its own context, the application can register its context with the context manager by calling **XDPSRegisterContext**. In either case, the application can then manipulate the context in a number of ways:

- Chain text contexts using **XDPSChainTextContext.**

- Set window system parameters for a context using **XDPSSetContextParameters**, or set individual parameters using **XDPSSetContextDepth**, **XDPSSetContextGrayMap**, **XDPSSetContextRGBMap**, or **XDPSSetContextDrawable**.

- Call **XDPSPushContextParameters** to temporarily set parameters and undo the results using **XDPSPopContextParameters**.

- Work with gstate objects (data structures that hold current graphics control parameters) by first capturing the current graphics state with **XDPSCaptureContextGState**, and then setting a context to the saved gstate object using **XDPSSetContextGState**. Use **XDPSPushContextGState** to temporarily set a context to a gstate object and later undo this action with **XDPSPopContextGState**. To update or to free a gstate object, **XDPSUpdateContextGState** and **XDPSFreeContextGState** can be called.

- Free contexts that are no longer needed by calling **XDPSDestroySharedContext**, which destroys a shared context and its space. **XDPSUnregisterContext** can be called to free context information without destroying the context.

3.2 Programming Tips

Capturing the current state with **XDPSCaptureContextGState** and restoring it later with **XDPSPushContextGState** or **XDPSSetContextGState** is more efficient than setting the parameters each time. However, each gstate object consumes memory, so don’t capture a gstate object unless you expect to return to it. You should also free gstate objects that are no longer being used, or recycle them with **XDPSUpdateContextGState**.

3.3 Procedure Overview

Table 2 *Context management procedures*

Procedure	*Functionality*
XDPSCaptureContextGState	Captures the current graphics state as a gstate object and returns a handle to it.
XDPSChainTextContext	Enables or disables a chained text context for a context.
XDPSDestroySharedContext	Destroys a shared context for a display and the context's space.
XDPSExtensionPresent	Determines whether a display supports the Display PostScript extension.
XDPSFreeContextGState	Releases a gstate object.
XDPSFreeDisplayInfo	Frees the stored display information for a display.
XDPSGetSharedContext	Returns the shared context for a display.
XDPSPopContextGState	Reverses the effects of **XDPSPushContextGState**.
XDPSPopContextParameters	Reverses the effects of **XDPSPushContextParameters**.
XDPSPushContextGState	Sets a context to a saved gstate object; can be undone by **XDPSPopContextGState**.
XDPSPushContextParameters	Sets context parameters; can be undone by **XDPSPopContextParameters**.
XDPSRegisterContext	Registers a context with the context manager.
XDPSSetContextDepth	Sets the screen and depth for a context.
XDPSSetContextDrawable	Sets the drawable for a context.
XDPSSetContextGrayMap	Sets the gray ramp for a context.
XDPSSetContextGState	Sets a context to a saved gstate object.
XDPSSetContextParameters	Sets context parameters.
XDPSSetContextRGBMap	Sets the RGB map for a context.
XDPSUnregisterContext	Frees context information for a context but doesn't destroy the context.
XDPSUpdateContextGState	Updates a saved gstate object to correspond to the current graphics state.

3.4 Structures

The *XDPSStandardColormap* structure is identical to the *XStandardColormap* structure but allows signed numbers for the multipliers.

XDPSStandardColormap

```
typedef struct {
 Colormap colormap;
 unsigned long red_max;
 long red_mult;
 unsigned long green_max;
 long green_mult;
 unsigned long blue_max;
 long blue_mult;
 unsigned long base_pixel;
 unsigned long visualid;
 unsigned long killid;
} XDPSStandardColormap;
```

The structure is used by **XDPSSetContextRGBMap**, **XDPSSetContextGrayMap, XDPSSetContextParameters**, and **XDPSPushContextParameters**.

3.5 Procedures

XDPSCaptureContextGState

```
int XDPSCaptureContextGState (context, *gsReturn)
 DPSContext context;
 DPSGState *gsReturn;
```

XDPSCaptureContextGState captures the current graphics state as a gstate object and returns a reference to it. *DPSGState* is an opaque type. It is legal to set a *DPSGState* variable to integer zero and to test it against zero—**XDPSCaptureContextGState** never returns zero in *gsReturn*.

XDPSCaptureContextGState returns *dps_status_unregistered_context* or *dps_status_success*.

XDPSChainTextContext

```
int XDPSChainTextContext (context, enable)
 DPSContext context;
 Bool enable;
```

XDPSChainTextContext either enables or disables a chained text context for a context. The first time **XDPSChainTextContext** is called with *enable* set to *True*, it creates the text context. The text context writes its output to the standard output file.

The context must have been registered with **XDPSRegisterContext**.

XDPSChainTextContext returns *dps_status_unregistered_context* or *dps_status_success*.

XDPSDestroySharedContext

```
void XDPSDestroySharedContext (context)
 DPSContext context;
```

XDPSDestroySharedContext destroys the shared context for a display; it also destroys the context's space.

XDPSExtensionPresent

```
Bool XDPSExtensionPresent (display)
 Display *display;
```

XDPSExtensionPresent returns *True* if *display* supports the Display PostScript extension, *False* otherwise.

XDPSFreeContextGState

```
int XDPSFreeContextGState (context, gs)
 DPSContext context;
 DPSGState gs;
```

XDPSFreeContextGState releases a gstate object previously acquired through **XDPSCaptureContextGState**.

XDPSFreeContextGState returns *dps_status_unregistered_context* or *dps_status_success.*

XDPSFreeDisplayInfo

```
void XDPSFreeDisplayInfo (display)
 Display *display;
```

XDPSFreeDisplayInfo frees the stored display information for *display.* It should be used if an application no longer needs to use the Display PostScript Toolkit on that display, but the application will be continuing.

XDPSGetSharedContext

```
DPSContext XDPSGetSharedContext (display)
 Display *display;
```

XDPSGetSharedContext returns the shared context for *display.* If no shared context exists, it creates one. **XDPSGetSharedContext** returns *NULL* if *display* does not support DPS/X.

The returned context is initially set to use the default colormap on the default screen with the default depth, but is not set to use any drawable.

XDPSPopContextGState

```
int XDPSPopContextGState (pushCookie)
 DPSPointer pushCookie;
```

XDPSPopContextGState restores a context to the state it was in before the call to **XDPSPushContextGState** that returned *pushCookie*.

XDPSPushContextGState and **XDPSPopContextGState** must be called in a stack-oriented fashion.

XDPSPopContextGState returns *dps_status_success* or *dps_status_illegal_value*.

XDPSPopContextParameters

```
int XDPSPopContextParameters (pushCookie)
 DPSPointer pushCookie;
```

XDPSPopContextParameters restores all context parameters to the state they were in before the call to **XDPSPushContextParameters** that returned *pushCookie*.

XDPSPushContextParameters and **XDPSPopContextParameters** must be called in a stack-oriented fashion.

XDPSPopContextParameters returns *dps_status_success* or *dps_status_illegal_value*.

XDPSPushContextGState

```
int XDPSPushContextGState (context, gs, pushCookieReturn)
 DPSContext context;
 DPSGState gs;
 DPSPointer *pushCookieReturn;
```

XDPSPushContextGState sets a context to a saved gstate object. This can be undone by passing the returned *pushCookieReturn* to **XDPSPopContextGState**.

XDPSPushContextGState and **XDPSPopContextGState** must be called in a stack-oriented fashion.

XDPSPushContextGState returns *dps_status_unregistered_context* or *dps_status_success.*

XDPSPushContextParameters

```
int XDPSPushContextParameters (context, screen, depth,
        drawable, height, rgbMap, grayMap, flags,
        pushCookieReturn)
 DPSContext context;
 Screen *screen;
 int depth;
 Drawable drawable;
 int height;
 XDPSStandardColormap *rgbMap;
 XDPSStandardColormap *grayMap;
 unsigned int flags;
 DPSPointer *pushCookieReturn;
```

XDPSPushContextParameters is identical to **XDPSSetContextParameters** but can be undone by passing the returned *pushCookieReturn* to **XDPSPopContextParameters**.

XDPSPushContextParameters and **XDPSPopContextParameters** must be called in a stack-oriented fashion.

XDPSPushContextParameters returns the same values as **XDPSSetContextParameters**.

XDPSRegisterContext

```
void XDPSRegisterContext (context, makeSharedContext)
 DPSContext context;
 Bool makeSharedContext;
```

XDPSRegisterContext registers a context with the context manager and makes it possible to manipulate the context using the procedures in this section.

If *makeSharedContext* is *True*, *context* becomes the shared context for the display. This does not destroy the previous shared context for the display, if there is one.

XDPSSetContextDepth

```
int XDPSSetContextDepth (context, screen, depth)
 DPSContext context;
 Screen *screen;
 int depth;
```

XDPSSetContextDepth sets a context for use with a particular screen and depth.

XDPSSetContextDepth returns *dps_status_unregistered_context* or returns *dps_status_success*. If *screen* is not on the context's display or *depth* is not valid for that screen, **XDPSSetContextDepth** returns *dps_status_illegal_value*.

XDPSSetContextDrawable

```
int XDPSSetContextDrawable (context, drawable, height)
 DPSContext context;
 Drawable drawable;
 int height;
```

XDPSSetContextDrawable sets a context for use with a particular drawable that has the specified *height*. The origin is at the lower left corner. The context must already be set for use with the drawable's screen; see **XDPSSetContextDepth**.

XDPSSetContextDrawable returns *dps_status_unregistered_context* or returns *dps_status_success*. If *height* is less than 1, **XDPSSetContextDrawable** returns *dps_status_illegal_value*.

XDPSSetContextGrayMap

```
int XDPSSetContextGrayMap (context, map)
 DPSContext context;
 XDPSStandardColormap *map;
```

XDPSSetContextGrayMap sets the gray ramp for *context*. The colormap in the *map* structure must be appropriate for the current drawable and depth. This colormap can be *None* when the context is imaging to a pixmap. In that case, the ramps must be set to the values used in the window that will display the pixmap.

If *map* is *NULL*, the default gray ramp for the default screen of the context's display is used. The *flags* parameter of **XDPSSetContextParameters** described below can be used to get the default for a nondefault screen.

The gray ramp is based upon the *base_pixel, red_max,* and *red_mult* fields of the *XDPSStandardColormap* structure; all other *_max* and *_mult* fields are ignored.

XDPSSetContextGrayMap returns *dps_status_unregistered_context* or returns *dps_status_success.*

XDPSSetContextGState

```
int XDPSSetContextGState (context, gs)
 DPSContext context;
 DPSGState gs;
```

XDPSSetContextGState sets a context to a saved gstate object. It returns *dps_status_success* or *dps_status_unregistered_context.*

XDPSSetContextParameters

```
int XDPSSetContextParameters (context, screen, depth,
        drawable, height, rgbMap, grayMap, flags)
 DPSContext context;
 Screen *screen;
 int depth;
 Drawable drawable;
 int height;
 XDPSStandardColormap *rgbMap;
 XDPSStandardColormap *grayMap;
 unsigned int flags;
```

XDPSSetContextParameters sets any of the context parameters. It uses the following macros to decide which parameters to set.

```
XDPSContextScreenDepth
XDPSContextDrawable
XDPSContextRGBMap
XDPSContextGrayMap
```

flags should be a bitwise *OR* of one or more of these values.

XDPSSetContextParameters returns *dps_status_success* if all requested changes were successfully made. If any parameter is in error, **XDPSSetContextParameters** returns either *dps_status_unregistered_context* or *dps_status_illegal_value* as appropriate. In the case of non-success, no changes were made.

If *flags* requires that a colormap is set and the corresponding *map* parameter is *NULL*, a default map is used. In that case:

- If *screen* is not *NULL*, the default map is the one set on the screen's root window.
- If *screen* is *NULL* and *drawable* is *None*, the default map is the one on the display's default root window.
- If *screen* is *NULL* but *drawable* is not *None*, the default map is the one set on the root window of the screen specified by *drawable*.

XDPSSetContextRGBMap

```
int XDPSSetContextRGBMap (context, map)
 DPSContext context;
 XDPSStandardColormap *map;
```

XDPSSetContextRGBMap sets the RGB color cube for *context*. The colormap in the *map* structure must be appropriate for the current drawable and depth. This colormap can be *None* if the application is rendering to a pixmap. In that case, the ramps must be set to the values used in the window that will display the pixmap.

If *map* is *NULL*, the default RGB cube for the default screen of the context's display is used. The *flags* parameter of **XDPSSetContextParameters** described above can be used to get the default for a nondefault screen.

XDPSSetContextRGBMap returns *dps_status_success* or returns *dps_status_unregistered_context.*

XDPSUnregisterContext

```
void XDPSUnregisterContext (context)
 DPSContext context;
```

XDPSUnregisterContext frees context information but doesn't destroy a context.

XDPSUpdateContextGState

```
int XDPSUpdateContextGState (context, gs)
 DPSContext context;
 DPSGState gs;
```

XDPSUpdateContextGState updates the saved gstate object to correspond to the current graphics state. The previous setting of the gstate object is no longer accessible.

XDPSUpdateContextGState returns *dps_status_unregistered_context* or *dps_status_success*.

4 User Objects

The toolkit procedures described in this section can be used to manage user objects such as user paths. These procedures are recommended for any DPS/X application that uses user objects. User objects are discussed in section 3.7.6 of *PostScript Language Reference Manual, Second Edition*.

The procedures documented in this section are compatible with those described in described in "User Object Indices" in section 3.3 of *Client Library Supplement for X*. An application can combine them as convenient.

Applications that call the procedures documented in this section must include *<DPS/dpsXshare.h>*. The procedures can be used for any context; it is not necessary to register the context first.

4.1 Procedure Overview

Two forms are provided for each user object management procedure, one starting with **DPS** and the other with **PS**. The procedures are identical, except for the first argument: the procedure starting with **PS** uses the current context, while the procedure starting with **DPS** requires a context as its first argument.

Table 3 *User object procedures*

Procedure	*Functionality*
PSDefineAsUserObj **DPSDefineAsUserObj**	Allocates a user object index and associates it with the item on top of the operand stack
PSRedefineUserObj **DPSRedefineUserObj**	Breaks the association between a user object index and its current object and associates the index with the item on top of the operand stack.
PSReserveUserObjIndices **DPSReserveUserObjIndices**	Reserves a number of user object indices for an application's use.
PSUndefineUserObj **DPSUndefineUserObj**	Breaks the association between a user object index and its current object.

4.2 Procedures

PSDefineAsUserObj
DPSDefineAsUserObj

```
int DPSDefineAsUserObj (context)
 DPSContext context;
```

DPSDefineAsUserObj allocates a user object index and associates it with the item on top of the operand stack. The return value is the user object index.

PSRedefineUserObj
DPSRedefineUserObj

```
void DPSRedefineUserObj (context, userObj)
 DPSContext context;
 int userObj;
```

DPSRedefineUserObj breaks the association between the user object index *userObj* and its current object. It then associates the index with the item on top of the operand stack.

PSReserveUserObjIndices
DPSReserveUserObjIndices

```
int DPSReserveUserObjIndices (context, number)
 DPSContext context;
 int number;
```

DPSReserveUserObjIndices reserves a specified number of user object indices for an application's use. It does not associate these indices with any objects. The return value is the first index reserved; if the return value is *f*, an application can freely use the indices *f* through *f+number–1*.

PSUndefineUserObj
DPSUndefineUserObj

```
void DPSUndefineUserObj (context, userObj)
 DPSContext context;
 int userObj;
```

DPSUndefineUserObj breaks the association between the user object index *userObj* and its current object. Further use of the index is not allowed. Future calls to **DPSDefineAsUserObj** might return the same index with a new association.

5 User Paths

The procedures described in this section provide convenient access to user paths. A *user path* is a PostScript language procedure that consists entirely of path construction operators and their coordinate operands expressed as literal numbers. User paths can also be expressed in a compact, encoded form. The compact form is the format generated by the procedures described in this section.

User paths are described in section 4.6 of *PostScript Language Reference Manual, Second Edition.* Applications that use the utilities described in this section must include <*DPS/dpsXuserpath.h*>.

5.1 Structures and Type Definitions

DPSNumberFormat

```
typedef enum _DPSNumberFormat {
 dps_float,
 dps_long,
 dps_short
} DPSNumberFormat;
```

DPSNumberFormat describes the format of numeric procedure call parameters.

- For floating point, 32-bit, or 16-bit values, use *dps_float*, *dps_long*, or *dps_short*, respectively.
- For 32-bit fixed-point numbers, use *dps_long* plus the number of bits in the fractional part.
- For 16-bit fixed-point numbers, use *dps_short* plus the number of bits in the fractional part.

Note: You cannot use 64-bit values with the procedures in this section.

TK

DPSUserPathOp

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

typedef char DPSUserPathOp;
```

DPSUserPathOp enumerates the PostScript operators that define a path.

DPSUserPathAction

```
typedef enum _DPSUserPathAction {
 dps_uappend,
 dps_ufill,
 dps_ueofill,
 dps_ustroke,
 dps_ustrokepath,
 dps_inufill,
 dps_inueofill,
 dps_inustroke,
 dps_infill,
 dps_ineofill,
 dps_instroke,
 dps_def,
 dps_put,
 dps_send
} DPSUserPathAction;
```

DPSUserPathAction enumerates the operators that can be applied to a path. The special action *dps_send* pushes the user path on the stack and leaves it there.

5.2 Procedure Overview

Two forms are provided for each user path procedure, one starting with **DPS** and the other with **PS**. The procedures are identical, except for the first argument: the procedure starting with **PS** uses the current context, while the procedure starting with **DPS** requires a context as its first argument.

A context used with these procedures does not need to be registered with the context management procedures described in section 3, “Context Management Procedures.”

Table 4 *User path procedures*

Procedure	*Functionality*
PSDoUserPath **DPSDoUserPath**	Sends a user path and operates on it, using the operator specified in the *action* parameter.
PSHitUserPath **DPSHitUserPath**	Sends a user path for one of the hit detection operators. The operator is specified in the *action* parameter.

5.3 Procedures

PSDoUserPath
DPSDoUserPath

```
void DPSDoUserPath (ctx, coords, numCoords, numType, ops,
          numOp, bbox, action)
 DPSContext ctx;
 DPSPointer coords;
 int numCoords;
 DPSNumberFormat numType;
 DPSUserPathOp *ops;
 int numOp;
 DPSPointer bbox;
 DPSUserPathAction action;
```

DPSDoUserPath provides a convenient interface to user paths.

coords is an array of coordinates for the operands. Do not include the parameters for the *dps_setbbox* operation in this array.

numCoords provides the number of entries in the *coords* array. The type of the entries in *coords* is defined by the *numType* parameter.

numType describes the number format used in the *coords* and *bbox* parameters.

ops points to an array of operations, as defined by *DPSUserPathOp*.

numOp gives the number of entries in the array pointed to by *ops*.

bbox points to four numbers in the format defined by *numType*.

action describes the PostScript operator that consumes the created user path.

The operator list in the *ops* parameter can, but need not, include a *dps_setbbox* operation. If *dps_setbbox* is not included, **DPSDoUserPath** inserts it at the appropriate place.

Each operator in the *ops* array consumes operands from the *coords* array. The number of coordinates varies for different operands, as shown in Table 5:

Table 5 *Operators and coordinates*

Operator	*# of Operands*	*Description*
dps_setbbox	none	see the *bbox* parameter
dps_moveto	2	*x, y*
dps_rmoveto	2	*dx, dy*
dps_lineto	2	*x, y*
dps_rlineto	2	*dx, dy*
dps_curveto	6	$x_1, y_1, x_2, y_2, x_3, y_3$
dps_rcurveto	6	$dx_1, dy_1, dx_2, dy_2, dx_3, dy_3$
dps_arc	5	x, y, r, ang_1, ang_2
dps_arcn	5	x, y, r, ang_1, ang_2
dps_arct	5	x_1, y_1, x_2, y_2, r
dps_closepath	none	
dps_ucache	none	

The following code fragment uses **DPSDoUserPath** to draw a 75-unit circle centered around the point (100,100) with a radius from (100, 100) to (175, 100):

```
static long coords[9] = {100, 100, 75, 0, 360,
                    100, 100, 75, 0};
static DPSUserPathOp ops[3] = {dps_arc, dps_moveto,
                    dps_rlineto};
static long bbox[4] = {25, 25, 175, 175};

DPSDoUserPath (ctxt, (DPSPointer) coords, 9, dps_long,
              ops, 3,
          (DPSPointer) bbox, dps_ustroke);
```

PSHitUserPath
DPSHitUserPath

```
Bool DPSHitUserPath (ctx, x, y, radius, coords, numCoords,
        numType, ops, numOp, bbox, action)
 DPSContext ctx;
 double x, y, radius;
 DPSPointer coords;
 int numCoords;
 DPSNumberFormat numType;
 DPSUserPathOp *ops;
 int numOp;
 DPSPointer bbox;
 DPSUserPathAction action;
```

DPSHitUserPath provides a convenient interface to PostScript operators that test for path intersection without actually painting anything. For more information, consult sections 4.5.3, “Insideness Testing,” and 7.3.2, “Hit Detection,” of *PostScript Language Reference Manual, Second Edition*.

If *radius* is zero, **DPSHitUserPath** uses the *x/y* form of the operator specified by *action*. If *radius* is nonzero, **DPSHitUserPath** constructs a circular user path centered on *x* and *y* with the specified radius and uses the aperture form of the specified action.

If *action* is *dps_ineofill*, *dps_infill*, or *dps_instroke*, **DPSHitUserPath** ignores the parameters specifying the user path and tests against the current path. If *action* is *dps_inueofill*, *dps_inufill*, or *dps_inustroke*, **DPSHitUserPath** uses the parameters specifying the user path to define the user path being tested against. If *action* is anything else, **DPSHitUserPath** returns *False* and does nothing else.

See **DPSDoUserPath** for a description of the *coords*, *numCoords*, *numType*, *ops*, *numOp*, and *bbox* parameters.

The procedure returns the resulting boolean value.

Note: *Calling* **DPSHitUserPath** *with radius zero and dps_ineofill, dps_infill, or dps_in-stroke as the action is semantically equivalent to calling the DPSineofill, DPSinfill, or* **DPSinstroke** *procedure.*

TK

6 File Preview Procedures

The procedures described in this section simplify rendering PostScript language files into X drawable objects (windows or pixmaps). Code that uses the procedures must include <*DPS/dpsXpreview.h*>, which automatically includes <*DPS/dpsXcommon.h*>.

The section starts with a brief introduction to the file preview utilities, followed by structure and type definitions, a procedure overview, and procedure definitions.

6.1 Introduction

The first step is optionally to call **XDPSSetFileFunctions** to supply file access procedures appropriate to the data source:

- **XDPSFileGetsFunc** and **XDPSFileRewindFunc** are the default procedures, suitable for a separate EPS file.
- **XDPSEmbeddedEPSFGetsFunc** and **XDPSEmbeddedEPSFRewindFunc** handle an EPSF section within a longer file.
- The application can also define its own procedures that mimic the behavior of **fgets** and **rewind**. In this case, the image source is not limited to files.

An application can render a file into a pixmap or a window. If the application renders an EPS file into a pixmap, it can use **XDPSCreatePixmapForEPSF** to create an appropriately sized pixmap. The *%%BoundingBox* comment in the EPS file and the *pixelsPerPoint* parameter to **XDPSCreatePixmapForEPSF** determine the size of the pixmap. **XDPSPixelsPerPoint** can be called for information about the resolution of the specified screen.

The application then calls **XDPSImageFileIntoDrawable** to actually render the file **XDPSImageFileIntoDrawable** can render a file into any X window or pixmap; it is not limited to pixmaps created by **XDPSCreatePixmapForEPSF**.

If the specified display does not support the Display PostScript extension, the image area is filled with a 50% gray stipple pattern, or filled with solid 1’s if the *createMask* argument to **XDPSImageFileIntoDrawable** is *True*.

If the display supports the Display PostScript extension, **XDPSImageFileIntoDrawable** starts executing the file, placing the resulting image into the drawable. The setting of *waitForCompletion* determines what happens next:

- If *waitForCompletion* is *True*, **XDPSImageFileIntoDrawable** waits until imaging is complete before it returns.

- If *waitForCompletion* is *False*, **XDPSImageFileIntoDrawable** waits for the amount of time specified by **XDPSSetImagingTimeout**. If imaging is not complete by this time, **XDPSImageFileIntoDrawable** returns *dps_status_imaging_incomplete*.

  If imaging was incomplete, **XDPSImageFileIntoDrawable** temporarily sets the imaging context's status handler so that the variable pointed to by *doneFlag* will become *True* when the imaging completes. The application must then call **XDPSCheckImagingResults** to find the results of imaging. *doneFlag* can only change its state as a result of handling a status event from the DPS/X server.

If **XDPSImageFileIntoDrawable** returns *dps_status_imaging_incomplete*, an application has to wait until **XDPSCheckImagingResults** returns a status that is not *dps_status_imaging_incomplete* before it does anything with the context. The context is otherwise left in an undefined state and imaging might not be correct.

When an application uses **XDPSImageFileIntoDrawable** with *waitForCompletion False*, using pass-through event delivery is highly recommended. There can otherwise be substantial delays between the time *doneFlag* is set and the time the application has an opportunity to test *doneFlag*. See "Event Dispatching" in section 4.8 of *Client Library Supplement for X* for more information.

An application can stop partial imaging by destroying the context with **DPSDestroySharedContext** if it is using the shared context, or with both **DPSDestroyContext** and **XDPSUnregisterContext** if it is not using the shared context.

While **XDPSCreatePixmapForEPSF** requires a correctly formed EPS file to find the bounding box, **XDPSImageFileIntoDrawable** can image any single-page PostScript language file into a drawable.

The following code example shows how to create a pixmap for an EPS file and image the file into that pixmap. This example assumes that *widget* is the widget that will ultimately display the image, *depth* is the depth of that widget, and *file* is the opened EPS file. In this example, the penultimate parameter to **XDPSImageFileIntoDrawable** is *True*, so **XDPSImageFileIntoDrawable** will not return until the imaging is complete.

Example 1 *Creating a pixmap and executing an EPS file*

C language code:

```
int status;
XRectangle bbox, pixelSize;
Pixmap p;
Bool doneFlag;
float pixelsPerPoint;
```

```
pixelsPerPoint = XDPSPixelsPerPoint(XtScreen(widget));

status = XDPSCreatePixmapForEPSF((DPSContext) NULL,
       XtScreen(widget), file, depth, pixelsPerPoint,
       &p, &pixelSize, &bbox);

switch (status) {
 case dps_status_success:
  break;
 case dps_status_failure:
  fprintf(stderr, "File is not EPSF\n");
  exit(1);
 case dps_status_no_extension:
  fprintf(stderr, "Server does not support DPS\n");
  exit(1);
 default:
  fprintf(stderr, "Internal error %d\n", status);
  exit(1);
}

status = XDPSImageFileIntoDrawable((DPSContext) NULL,
       XtScreen(widget), p, file, pixelSize.height,
       depth, &bbox, -bbox.x, -bbox.y, pixelsPerPoint,
       True, False, True, &doneFlag);

switch (status) {
 case dps_status_success:
  break;
 case dps_status_no_extension:
  fprintf(stderr, "Server does not support DPS\n");
  exit(1);


 case dps_status_postscript_error:
  fprintf(stderr,
         "PostScript execution error in EPSF file\n");
  exit(1);
 default:
  fprintf(stderr, "Internal error %d\n", status);
  exit(1);
}
```

An EPS file can take a long time to execute. Worse, a poorly written EPS file might contain an infinite loop in its PostScript language code and never finish executing. One way to protect an application that imports EPS files is to use **XDPSImageFileIntoDrawable** with the *waitForCompletion* parameter *False* and allow the user to abort execution. The following code example contains the framework for doing this.

The *waitForCompletion* parameter to **XDPSImageFileIntoDrawable** (located next to last in the parameter list) has the value *False*, so the procedure call can return before the imaging is complete. If **XDPSImageFileIntoDrawable** returns *dps_status_imaging_incomplete*, the example goes into a subsidiary event dispatching loop until *doneFlag* becomes *True*. An application that gives the user a way to abort the execution of the EPS file would add an additional exit criterion to the dispatching loop. The example below assumes that the application has already set up pass-through event dispatching with **XDPSSetEventDelivery**.

Example 2 *Protecting against incorrect EPS files*

```
status = XDPSImageFileIntoDrawable((DPSContext) NULL,
       XtScreen(widget), p, file, pixelSize.height,
       depth, &bbox, –bbox.x, -bbox.y, pixelsPerPoint,
       True, False, False, &doneFlag);

if (status == dps_status_imaging_incomplete) {
 XEvent ev;
 do {
   XtAppNextEvent(app, &ev);
   if (!XDPSDispatchEvent(&ev)) XtDispatchEvent(&ev);
 } while (!doneFlag);
 status = XDPSCheckImagingResults((DPSContext) NULL,
       XtScreen(shell));
}

switch (status) {

/* ... as before ... */

}
```

6.2 Structures and Type Definitions

XDPSGetsFunction

```
typedef char *(*XDPSGetsFunction) (/*
 char *buf,
 int n,
 FILE *f,
 DPSPointer private*/);
```

XDPSGetsFunction is a procedure type. An *XDPSGetsFunction* mimics the behavior of the standard C library **fgets** procedure and returns the next line of a specified file. The *XDPSGetsFunction* returns *NULL* to indicate the end of the section to be imaged.

XDPSPosition

```
typedef struct {
 long startPos;
 int nestingLevel;
 unsigned long binaryCount;
 Bool continuedLine;
} XDPSPosition;
```

This data structure is used with **XDPSEmbeddedEPSFRewindFunc** and **XDPSEmbeddedEPSFGetsFunc** and is described there.

XDPSRewindFunction

```
typedef void (*XDPSRewindFunction) (/*
 FILE *f,
 DPSPointer private */);
```

XDPSRewindFunction is a procedure type. An *XDPSRewindFunction* mimics the standard C library **rewind** procedure and repositions the specified file to the beginning of the section to be imaged, normally with **fseek**. When **XDPSImageFileIntoDrawable** and **XDPSCreatePixmapForEPSF** start to read lines from a file, they first execute the *XDPSRewindFunction* procedure.

6.3 Procedure Overview

Table 6 *File preview procedures*

Procedure	*Functionality*
XDPSCheckImagingResults	Checks the status of the imaging on a specified context.
XDPSCreatePixmapForEPSF	Creates a pixmap for imaging on a specified screen.
XDPSEmbeddedEPSFRewindFunc **XDPSEmbeddedEPSFGetsFunc**	These are **rewind** and **gets** procedures that handle an EPSF section embedded within a longer file.
XDPSFileRewindFunc **XDPSFileGetsFunc**	These are the default **rewind** and **gets** procedures that handle a separate EPS file.
XDPSImageFileIntoDrawable	Images a PostScript language file into a specified drawable.
XDPSPixelsPerPoint	Returns the resolution of a specified screen in pixels per point.
XDPSSetFileFunctions	Defines the procedures used by **XDPSCreatePixmapForEPSF** and **XDPSImageFileIntoDrawable** to reset a file to its beginning and to read the next line of the file.
XDPSSetImagingTimeout	Determines how long, in milliseconds, **XDPSImageFileIntoDrawable** waits before returning after incomplete imaging.

6.4 Procedures

XDPSCheckImagingResults

```
int XDPSCheckImagingResults (context, screen)
 DPSContext context;
 Screen *screen;
```

XDPSCheckImagingResults checks the status of the imaging on *context.*

If *context* is *NULL*, the shared context for *screen*'s display is used. If a non-*NULL* context is passed, it must have been registered with **XDPSRegisterContext**.

XDPSCheckImagingResults returns:

- *dps_status_success* if imaging is complete and successful.
- *dps_status_imaging_incomplete* if imaging is continuing.
- *dps_status_postscript_error* if imaging is complete but the PostScript language file being executed contains an error.
- *dps_status_illegal_value* if the context is not currently involved in previewing.
- *dps_status_unregistered_context* if the context has not been registered with the context manager.

XDPSCreatePixmapForEPSF

```
int XDPSCreatePixmapForEPSF (context, screen, epsf, depth,
                             pixelsPerPoint, pixmapReturn,
                             pixelSizeReturn, bboxReturn)
DPSContext context;
Screen *screen;
FILE *epsf;
int depth;
double pixelsPerPoint;
Pixmap *pixmapReturn;
XRectangle *pixelSizeReturn;
XRectangle *bboxReturn;
```

XDPSCreatePixmapForEPSF creates a pixmap for use on the specified screen. The *%%BoundingBox* comment in the file, scaled by *pixelsPerPoint*, determines the size of the pixmap.

context can be *NULL*. In that case, the shared context for *screen*'s display is used. If *context* is non-*NULL*, it must have been registered with **XDPSRegisterContext**.

XDPSCreatePixmapForEPSF returns one of the status values shown in Table 7.

Table 7 *Status return values for XDPSCreatePixmapForEPSF*

Status	*Description*
dps_status_success	This value is returned when **XDPSCreatePixmapForEPSF** completes successfully.
dps_status_no_extension	If this value is returned, the procedure still creates a pixmap and returns a suitable size. However, **XDPSImageFileIntoDrawable** will not be able to image to the pixmap since the Display PostScript extension is not present.
dps_status_illegal_value	This status value is returned if *screen* is *NULL*, *file* is *NULL*, or *depth* or *pixelsPerPoint* is less than or equal to 0.
dps_status_failure	This status value is returned if the file specified by *epsf* does not contain a *%%BoundingBox* comment.

XDPSCreatePixmapForEPSF returns the size of the pixmap in *pixelSizeReturn* (*x* and *y* are zero) and the bounding box (in points) in *bboxReturn*.

XDPSEmbeddedEPSFGetsFunc

```
extern char *XDPSEmbeddedEPSFGetsFunc (buf, n, f, data)
 char *buf;
 int n;
 FILE *f;
 DPSPointer data;
```

XDPSEmbeddedEPSFRewindFunc

```
extern void XDPSEmbeddedEPSFRewindFunc (f, data)
 FILE *f;
 DPSPointer data;
```

XDPSEmbeddedEPSFRewindFunc and **XDPSEmbeddedEPSFGetsFunc** are **rewind** and **gets** procedures that handle an EPS file embedded within a longer file. To preview a separate EPS file, use **XDPSFileRewindFunc** and **XDPSFileGetsFunc.**

An application can pass the **rewind** and **gets** procedures to **XDPSSetFileFunctions**. The *rewindPrivateData* and *getsPrivateData* arguments to **XDPSSetFileFunctions** must both point to the same instance of an *XDPSPosition* structure.

The procedures use the document structuring conventions comments *%%BeginDocument* and *%%EndDocument* (DSC version 2.0 or later) to detect the end of the included file and to identify any subsidiary EPSF sections included in the EPSF section being executed.

The application must set the *startPos* in the *XDPSPosition* structure to the first character of the desired EPSF section before calling **XDPSCreatePixmapForEPSF** or **XDPSImageFileIntoDrawable**. The position must be *after* any initial *%%BeginDocument* comment for this EPSF section.

The *nestingLevel*, *continuedLine*, and *binaryCount* fields are used internally by the procedures and should not be modified. A call to **XDPSImageFileIntoDrawable** modifies *startPos* to be the first character after the complete EPSF section, or –1 if the EPSF section ended with end-of-file.

XDPSFileGetsFunc

```
extern char *XDPSFileGetsFunc (buf, n, f, private)
 char *buf;
 int n;
 FILE *f;
 DPSPointer private;
```

XDPSFileRewindFunc

```
extern void XDPSFileRewindFunc (f, private)
 FILE *f;
 DPSPointer private;
```

XDPSFileGetsFunc and **XDPSFileRewindFunc** are the default **gets** and **rewind** procedures and are appropriate for an EPSF file that is a separate file. Use **XDPSEmbeddedEPSFRewindFunc** and **XDPSEmbeddedEPSFGetsFunc** while previewing an EPSF section embedded in a longer file.

If an application has installed different procedures for this behavior, **XDPSFileRewindFunc** or **XDPSFileGetsFunc** can be passed to **XDPSSetFileFunctions** to restore the default behavior. The *rewindPrivateData* and *getsPrivateData* pointers should both be *NULL*.

XDPSImageFileIntoDrawable

```
extern int XDPSImageFileIntoDrawable (context,
    screen, dest, file, drawableHeight, drawableDepth,
    bbox, xOffset, yOffset, pixelsPerPoint, clear,
    createMask, waitForCompletion, doneFlag)
 DPSContext context;
 Screen *screen;
 Drawable dest;
 FILE *file;
 int drawableHeight, drawableDepth;
 XRectangle *bbox;
 int xOffset, yOffset;
 double pixelsPerPoint;
 Bool clear, createMask, waitForCompletion, *doneFlag;
```

XDPSImageFileIntoDrawable images a PostScript language file into the *dest* drawable object—that is, into a pixmap or a window.

If *context* is *NULL*, the shared context for the display is used. If a context is passed, it must have been registered with **XDPSRegisterContext**.

drawableHeight and *drawableDepth* describe the drawable object; the height is in X pixels.

bbox describes the bounding box of the imaged area, in points.

The image is offset by *xOffset* and *yOffset*, which are given in points. The offsets are often *–bbox.x* and *–bbox.y*, which shifts the image to the lower left corner of the drawable.

pixelsPerPoint defines the scale factor used to image the PostScript language file.

If *clear* is *True*, the area defined by *bbox* is cleared to white before imaging.

If *createMask* is *True*, the drawable must be 1 bit deep, and becomes a mask that can be used as an X clip mask: each bit that the PostScript interpreter touches during imaging is set to 1. If *clear* is also *True*, all untouched bits within *bbox* are set to 0.

If *waitForCompletion* is *True*, **XDPSImageFileIntoDrawable** waits until imaging is complete before returning. If *waitForCompletion* is *False*, **XDPSImageFileIntoDrawable** waits for the amount of time specified by **XDPSSetImagingTimeout** and then returns *dps_status_imaging_incomplete* if imaging is not complete.

When imaging is complete, agents set up by **XDPSImageFileIntoDrawable** set the variable pointed to by *doneFlag* to *True*. The application must then call **XDPSCheckImagingResults** to find the results of imaging. The status of *doneFlag* can only change as a result of handling a status event from the DPS/X server.

Incorrect imaging can result, and a context can be left in an undefined state, if anything is done to affect the context between the following times:

- When **XDPSImageFileIntoDrawable** returns *dps_status_imaging_incomplete*
- When **XDPSCheckImagingResults** returns a status that is not *dps_status_imaging_incomplete*

To cancel imaging, the application can destroy the context by calling **DPSDestroySharedContext** or by calling both **DPSDestroyContext** and **XDPSUnregisterContext**.

When an application uses **XDPSImageFileIntoDrawable** with *waitForCompletion False*, using pass-through event delivery is highly recommended. There can otherwise be substantial delays between the time *doneFlag* is set and the time the application gets the opportunity to test *doneFlag*. See *Client Library Supplement for X*, section 5.3, "Use Pass-Through Event Dispatching," for more information.

If a display does not support the Display PostScript extension, the image area determined by the *bbox* parameter is filled with a 50% gray stipple pattern, or is filled with solid 1's if *createMask* is *True*.

XDPSImageFileIntoDrawable returns *dps_status_success*, *dps_status_no_extension*, or *dps_status_unregistered_context*, or one of the following values:

- *dps_status_illegal_value* if *screen* is *NULL*, *drawable* is *None, file* is *NULL*, or *drawableHeight*, *drawableDepth*, or *pixelsPerPoint* is less than or equal to 0.
- *dps_status_postscript_error* if the PostScript language file contains an error.
- *dps_status_imaging_incomplete* if *waitForCompletion* is *False* and the imaging is not finished within the time-out.

XDPSPixelsPerPoint

```
extern double XDPSPixelsPerPoint (screen)
 Screen *screen;
```

XDPSPixelsPerPoint returns the resolution of *screen*; this value can be passed to **XDPSCreatePixmapForEPSF** or **XDPSImageFileIntoDrawable**.

Note: If the X server reports incorrect resolution information about the screen, as is the case in some implementations, the incorrect information is propagated by **XDPSPixelsPerPoint**.

XDPSSetFileFunctions

```
extern int XDPSSetFileFunctions (rewindFunction,
    rewindPrivateData, getsFunction, getsPrivateData)
 XDPSRewindFunction rewindFunction;
 DPSPointer rewindPrivateData;
 XDPSGetsFunction getsFunction;
 DPSPointer getsPrivateData;
```

XDPSSetFileFunctions defines the procedures that **XDPSCreatePixmapForEPSF** and **XDPSImageFileIntoDrawable** use to reset a file to its beginning and to read the next line of the PostScript language file.

The values specified by *rewindPrivateData* and *getsPrivateData* are passed as the *private* parameter to the *rewind* and *gets* procedures, but are otherwise ignored.

The default procedures are suitable for use with a file that contains a single EPSF image. They can be replaced with procedures to handle, for example, an EPSF section embedded within a longer file.

XDPSSetImagingTimeout

```
extern void XDPSSetImagingTimeout (timeout, maxDoublings)
 int timeout, maxDoublings;
```

XDPSSetImagingTimeout determines how long (in milliseconds) **XDPSImageFileIntoDrawable** waits before returning that imaging incomplete. **XDPSImageFileIntoDrawable** first waits for the amount of time specified by *timeout* and then repeatedly doubles the wait until imaging is complete or until *maxDoublings* have occurred.

7 The Motif Font Selection Panel

The font selection panel is a Motif dialog box that allows the end user to choose one of the available Type 1 fonts. It presents the fonts available on the workstation and any fonts that can be located through the *PSRESOURCEPATH* environment variable. (See Appendix A, “Locating PostScript Language Resources.”) The user can choose a font by selecting the font family, face, and size, then view the font in the preview window above the selection panels.

From the font selection panel, the user can bring up a font sampler (see section 8, “The Motif Font Sampler”). The font sampler makes it possible to view fonts with certain characteristics—for example, to view all currently available bold italic fonts.

The following sections provide information on the font selection panel and the font sampler, including

- The behavior of the font selection panel and the font sampler.
- The available resources for the font selection panel and the font sampler.
- Callback procedures and associated callback information.
- Procedures for working with the font selection panel and the font sampler.

Note: An application that creates a font selection panel must merge the contents of the FontSelect defaults file into its own application defaults file. Beginning with the X11R5 release of the X Window System, this can be done with a #include directive in the application defaults file.

An application normally creates the font selection panel as a child of a shell widget, usually a transient shell. The font selection panel can also be elsewhere in the widget hierarchy. This allows the application to put additional information around the font selection panel. In that case, the application is responsible for popping up and popping down the Font Selection Panel.

The following information lets you use the widget:

- The header file is *<DPS/FontSB.h>*.
- The class pointer is *fontSelectionBoxWidgetClass*.
- The class name is *FontSelectionBox*.
- The *FontSelectionBox* widget is a subclass of *XmManager*.

7.1 Using the Motif Font Selection Panel

This section describes the behavior of the font selection panel in more detail. Information about the resources, callbacks, and procedures that implement the behavior are documented in the following sections. An example of a font selection panel is shown in Figure 1.

Figure 1 *The font selection panel*

7.1.1 Introduction

At the top of the font selection panel, a display region shows the selected font. This region, which is as wide as the panel, is called the *preview window.* The user can resize the preview window by moving the square handle at the lower right corner of the preview window. Below the preview window, the Family list region on the left and the Face list region on the right show the available fonts. Each time the user chooses a font family from the Family list, the Face list is updated appropriately. For example, a Face list for Helvetica might include Bold and Oblique, while a Face list for New Caledonia might include Bold and Italic.

Below the Family list region are a type-in region for selecting a font size and an option button. Below the Face list region are the Sampler button that brings up the font sampler and the Preview button. At the bottom of the font selection panel, the user can choose the OK, Apply, Reset, or Cancel buttons to apply or undo the selection.

7.1.2 The Sampler Button

When the user activates the Sampler button, the font selection panel creates and displays a font sampler as described in Section 8.

7.1.3 The Preview Button

When the user activates the Preview button, the preview window displays the currently selected font name in that font. Typing *p* or *P* into the *size* text field or double-clicking in the Family or Face list is equivalent to activating the Preview button. Previewing can be made automatic with the *XtNautoPreview* resource. See Table 8 for the font selection panel resource set.

7.1.4 The OK Button

When the user activates the OK button, the font selected in the panel is returned to the application and the font selection panel disappears, as described below.

1. Any fonts downloaded for preview which do not correspond to the current selection are undefined.

2. The panel looks for the name of the selected font's AFM (Adobe Font Metric) file if the *XtNgetAFM* resource is *True* and the current settings are for exactly one font.

3. *XtNvalidateCallback* is invoked with the current settings in the panel. *FSBCallbackReason* is *FSBOK*.

- If the *doit* field in the call data is now *False*, the panel does nothing more and remains on screen without calling *XtNokCallback*.

- Otherwise, the panel uses the current selections to update the resources *XtNfontName, XtNfontSize, XtNfontFamily, XtNfontFace, XtNfontNameMultiple, XtNfontFamilyMultiple, XtNfontFaceMultiple*, and *XtNfontSizeMultiple* with the current selections.

4. *XtNokCallback* is called with the current settings. *FSBCallbackReason* is *FSBOK*.

5. If the parent of the font selection panel is a shell, the panel pops down the shell.

Note: If the parent is not a shell, the application should make the font selection panel disappear in its XtNokCallback.

7.1.5 The Apply Button

When the user activates the Apply button, the font selection panel performs all the operations for the OK button but does not pop down the panel's parent shell. *XtNapplyCallback* is called instead of *XtNokCallback*. *FSBCallbackReason* is *FSBApply* in all callbacks.

7.1.6 The Reset Button

When the user activates the Reset button, the selected font reverts to the one selected when the user last chose Apply or OK, or the one last set by the application, whichever happened most recently.

To accomplish this, the font selection panel performs the following actions:

- First, the panel restores the current settings to those specified by the resources *XtNfontName, XtNfontFamily, XtNfontFace, XtNfontSize, XtNfontNameMultiple, XtNfontFamilyMultiple, XtNfontFaceMultiple*, and *XtNfontSizeMultiple*.

- Then all fonts which were downloaded for preview, but which do not correspond to the current settings, are undefined.

- After that, the panel calls *XtNresetCallback* with the current settings. The settings are identical to those passed to the most recent invocation of *XtNokCallback* or *XtNapplyCallback*, or to the most recent settings specified by the application, whichever happened last. *FSBCallbackReason* is *FSBReset*.

7.1.7 The Cancel Button

When the user activates the Cancel button, the font selection panel performs all operations listed for the Reset button, but calls *XtNcancelCallback* instead of *XtNresetCallback*. *FSBCallbackReason* is *FSBCancel*. If the parent of the font selection panel is a shell, the panel pops down the shell.

Note: If the parent is not a shell, the application should make the font selection panel disappear in its XtNcancelCallback.

7.2 Application Control of the Font Panel

The application can set the currently selected font in the font selection panel. It does this either by specifying a font name (for example, "Helvetica-BoldOblique") for the *XtNfontName* resource or by specifying a font family and face (for example "Helvetica" and "Bold Oblique") for the *XtNfontFamily* and *XtNfontFace* resources. The boolean resource *XtNuseFontName* controls whether the font selection panel pays attention to the

font name resource or the font family and face resources. The two interface procedures *FSBSetFontName* and *FSBSetFontFamilyFace* provide convenient interfaces to these resources.

The currently selected font size can be set with the *XtNfontSize* resource. This is a floating point resource, and is therefore difficult to set with *XtSetValues*. The interface procedure *FSBSetFontSize* provides the same functionality and is easier to use.

The application can also tell the font selection panel to display the fact that multiple fonts or sizes are currently selected. The boolean resources *XtNfontNameMultiple*, *XtNfontFamilyMultiple*, *XtNfontFaceMultiple*, and *XtNfontSizeMultiple* control this; there are also parameters to the convenience interface procedures that set these resources.

Setting multiple fonts allows some useful interaction techniques. For example, assume that the user has selected a block of text that contains several different fonts. The application sets a multiple font selection in the font selection panel.

- If the user selects a new size but makes no font selection, the application can make all the text in the block the selected size without changing the fonts.

- If the user selects a new font family but not a new font face, the application can convert each face in the block to the corresponding face in the new family by calling *FSBMatchFontFace*.

The callback data passed to the application indicates when there is a multiple font or size selection. A multiple font or size selection can result only from the application's setting a multiple selection that the user does not subsequently change; the user cannot convert a nonmultiple selection into a multiple selection.

7.3 Font Downloading and Resource Database Files

Each implementation of the Display PostScript extension has a directory or set of directories where it looks for Type 1 font outline programs. The fonts described in these programs are the fonts that appear in the font selection panel. The font selection panel can also temporarily download other font programs into the Display PostScript extension.

Generally, users should install new fonts in the normal font outline directory. However, there can be reasons why a user cannot or does not want to do this:

- The user might not have permission to add files to the outline directory.

- The font program might be the user's own private copy, and the outline directory might be shared among different machines.

- The user might have so many font programs available, for example on a file server, that unsophisticated programs would bog down if all the fonts were installed in the outline directory.

The font selection panel uses the *PSRESOURCEPATH* environment variable to locate fonts to download. This environment variable lists directories that contain PostScript language resource database files, and the resource database files in turn list the names of files that contain font programs to download. Appendix A gives full details of these resource database files. When the user chooses a font that is not resident in the server, the font selection panel automatically downloads the font into the Display PostScript extension. This is somewhat slower than using a resident font, but it is otherwise transparent to the user.

In addition to names of font programs, resource database files contain name information about each font, whether it is downloadable or resident in the server. This scheme allows the font selection panel to list the font without having to look into the font program itself. When no name information is found for a resident font, the font selection panel queries the server for this information, but this query takes much longer than fetching the information from resource database files—up to 100 times longer.

For efficient performance, always be sure to provide resource database files for resident fonts. If creating a font selection panel takes a long time, the reason is probably that resource database files are not available.

7.4 Font Selection Resources

Table 8 *Motif font selection panel resource set*

Name	*Class*	*Default*	*Type*	*Access*
XtNautoPreview	XmCAutoPreview	*True*	XtRBoolean	CSG
XtNcontext	XmCContext	*NULL*	XtRDPSContext	CSG
XtNdefaultResourcePath	XmCDefaultResourcePath	*See description*	XtRString	CSG
XtNfontFace	XmCFontFace	*NULL*	XtRString	CSG
XtNfontFaceMultiple	XmCFontFaceMultiple	*False*	XtRBoolean	CSG
XtNfontFamily	XmCFontFamily	*NULL*	XtRString	CSG
XtNfontFamilyMultiple	XmCFontFamilyMultiple	*False*	XtRBoolean	CSG
XtNfontName	XmCFontName	*NULL*	XtRString	CSG
XtNfontNameMultiple	XmCFontNameMultiple	*False*	XtRBoolean	CSG

Table 8 *Motif font selection panel resource set (Continued)*

Name	*Class*	*Default*	*Type*	*Access*
XtNfontSize	XmCFontSize	12.0	XtRFloat	CSG
XtNfontSizeMultiple	XmCFontSizeMultiple	*False*	XtRBoolean	CSG
XtNgetAFM	XmCGetAFM	*False*	XtRBoolean	CSG
XtNgetServerFonts	XmCGetServerFonts	*True*	XtRBoolean	CSG
XtNmakeFontsShared	XmCMakeFontsShared	*True*	XtRBoolean	CSG
XtNmaxPendingDeletes	XmCMaxPendingDeletes	10	XtRInt	CSG
XtNpreviewOnChange	XmCPreviewOnChange	*True*	XtRBoolean	CSG
XtNpreviewString	XmCPreviewString	*NULL*	XtRString	CSG
XtNresourcePathOverride	XmCResourcePathOverride	*NULL*	XtRString	CSG
XtNsizeCount	XmCSizeCount	10	XtRInt	CSG
XtNsizes	XmCSizes	*See description*	XtRFloatList	CSG
XtNshowSampler	XmCShowSampler	*False*	XtRBoolean	CSG
XtNshowSamplerButton	XmCShowSamplerButton	*True*	XtRBoolean	CSG
XtNundefUnusedFonts	XmCUndefUnusedFonts	*True*	XtRBoolean	CSG
XtNuseFontName	XmCUseFontName	*True*	XtRBoolean	CSG

7.4.1 Resource Description

XtNautoPreview If *True*, the font selection panel previews fonts as soon as the user selects them. If *False*, the user must activate the Preview button. Default is *True.*

XtNcontext Provides a context to use for previewing and querying the server for fonts. The font selection panel changes the *drawable* and possibly the *depth* for this context. If the context is *NULL*, the panel uses the shared context for the display. Default is *NULL*.

XtNdefaultResourcePath Provides the default *path* parameter for locating AFM files and fonts that can be downloaded. Default is specified at compilation time. See Appendix A, “Locating PostScript Language Resources,” for more information.

XtNfontFace Provides the selected face name. Relevant only if *XtNuseFontName* is *False*. If *NULL*, the face is selected by the *XtNfaceSelectCallback* resource. Default is *NULL*.

XtNfontFaceMultiple If *True*, displays a message that multiple faces are selected. Default is *False*.

XtNfontFamily Provides the selected font family. Relevant only if *XtNuseFontName* is *False.* If *NULL*, no family is selected. Default is *NULL*.

XtNfontFamilyMultiple If *True*, displays a message that multiple families are selected. Default is *False.*

XtNfontName Provides the selected font name. Relevant only if *XtNuseFontName* is *True*. If *NULL*, no font is selected. Default is *NULL*.

XtNfontNameMultiple If *True,* displays a message that multiple families and faces are selected. Default is *False*.

XtNfontSize Provides selected font size. Default is 12.0. Setting this resource with **XtSetValues** is difficult. Use **FSBSetFontSize** instead.

XtNfontSizeMultiple If *True*, indicates that multiple sizes are selected. Default is *False*.

XtNgetAFM If *True*, the font selection panel tries to find an AFM file before calling *XtNokCallback* or *XtNapplyCallback*. Default is *False*.

XtNgetServerFonts If *True*, list both resident and downloadable fonts. If *False*, list only downloadable fonts. Default is *True*.

XtNmakeFontsShared *XtNmakeFontsShared* and *XtNundefUnusedFonts* control where fonts are defined and whether the font selection panel undefines fonts that were previewed but not selected.

The possible font selection panel behaviors for values of *XtNundefUnusedFonts* and *XtNmakeFontsShared* are shown in Table 9.

TK

Table 9 *Behaviors for XtNundefUnusedFonts and XtNmakeFontsShared*

XtNundefUnusedFonts	*XtNmakeFontsShared*	*Behavior*
False	*False*	The panel loads the fonts into private VM and never undefines fonts.
False	*True*	The panel loads the fonts into shared VM and never undefines fonts.
True	*False*	The panel loads the fonts into private VM and undefines unused fonts when the user activates the OK, Apply, Reset, or Cancel buttons, or when there are more unused fonts than specified in the *XtNmaxPendingDeletes* resource.
True	*True*	The panel loads the fonts into private VM. When the user activates the OK or Apply button, all fonts downloaded into private VM are undefined and the selected font is downloaded into shared VM. Fonts are also undefined if the user activates the Reset or Cancel button, or when there are more unused fonts than specified in the XtNmaxPendingDeletes resource.

If *XtNmakeFontsShared* is *False*, the application must use the same context as the font selection panel, otherwise loaded fonts will not be available to the application.

Default is *True*.

XtNmaxPendingDeletes If *XtNundefUnusedFonts* is *True*, *XtNmaxPendingDeletes* specifies the maximum number of unused fonts allowed to remain before the font selection panel undefines the least recently loaded font. Making this value too small leads to repeated downloading during typical browsing. Making this value too large leads to excessive server memory use. Default is 10.

XtNpreviewOnChange If *XtNautoPreview* is *False*, *XtNpreviewOnChange* controls whether the font selection panel preview changes when the application changes *fontName*, *fontFamilyName*, *fontFaceName*, or *fontSize*. Default is *True*.

XtNpreviewString Determines the string displayed in the preview window. If *NULL*, displays the font name. Default is *NULL*.

XtNresourcePathOverride If non-*NULL*, provides a resource path to override the user's *PSRESOURCEPATH* environment variable. Default is *NULL*.

XtNsizeCount Determines the number of entries in the *XtNsizes* resource. Default is 10.

XtNsizes Provides a list of sizes to present in the Size menu. Default is 8, 10, 12, 14, 16, 18, 24, 36, 48, 72.

XtNshowSampler Determines whether the font sampler is shown when the font selection panel pops up. Tracks the popped-up state of the sampler, and can be used to pop the sampler up or down. Default is *False.*

XtNshowSamplerButton Determines whether or not the button to bring up the font sampler is visible. Default is *True.*

XtNundefUnusedFonts Default is *True.* A description is given under *XtNmakeFontsShared.*

XtNuseFontName Determines whether the *XtNfontName* or the *XtNfontFamily* and *XtNfontFace* resources are used to choose the initial font to display. Default is *True.*

7.4.2 Children of the Motif Font Panel

The following resources provide access to the child widgets of the font selection panel. They cannot be changed.

The name of each child widget is the same as the resource name, but without the *Child* suffix.

Table 10 *Motif font selection panel child resource set*

Name	*Class*	*Type*	*Access*
XtNapplyButtonChild	XtCReadOnly	XtRWidget	G
XtNcancelButtonChild	XtCReadOnly	XtRWidget	G
XtNfaceLabelChild	XtCReadOnly	XtRWidget	G
XtNfaceMultipleLabelChild	XtCReadOnly	XtRWidget	G
XtNfaceScrolledListChild	XtCReadOnly	XtRWidget	G
XtNfamilyMultipleLabelChild	XtCReadOnly	XtRWidget	G
XtNfamilyScrolledListChild	XtCReadOnly	XtRWidget	G
XtNokButtonChild	XtCReadOnly	XtRWidget	G
XtNpaneChild	XtCReadOnly	XtRWidget	G
XtNpanelChild	XtCReadOnly	XtRWidget	G
XtNpreviewButtonChild	XtCReadOnly	XtRWidget	G
XtNpreviewChild	XtCReadOnly	XtRWidget	G
XtNresetButtonChild	XtCReadOnly	XtRWidget	G
XtNsamplerButtonChild	XtCReadOnly	XtRWidget	G
XtNseparatorChild	XtCReadOnly	XtRWidget	G

Table 10 *Motif font selection panel child resource set (Continued)*

Name	*Class*	*Type*	*Access*
XtNsizeLabelChild	XtCReadOnly	XtRWidget	G
XtNsizeMultipleLabelChild	XtCReadOnly	XtRWidget	G
XtNsizeOptionMenuChild	XtCReadOnly	XtRWidget	G
XtNsizeTextFieldChild	XtCReadOnly	XtRWidget	G

7.5 Callback Procedures

The following sections contain information about callback procedures available for working with the font selection panel. The resource table is followed by a short description of each callback.

Table 11 *Motif font selection panel callback resource set*

Name	*Class*	*Default*	*Type*	*Access*
XtNapplyCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNcancelCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNcreateSamplerCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNfaceSelectCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNokCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNresetCallback	XtCCallback	*NULL*	XtCallbackList	C
XtNvalidateCallback	XtCCallback	*NULL*	XtCallbackList	C

XtNapplyCallback Indicates that the user wants to choose the selected options. The font selection panel remains. *XtNapplyCallback* passes a pointer to an *FSBCallbackRec* as call data. Applications typically supply the same procedure for the *XtNapplyCallback* as for *XtNokCallback*.

XtNcancelCallback The font selection panel reverts to the values last set with *XtNfontName*, *XtNfontFamily*, *XtNfontFace*, and *XtNfontSize*. The font selection panel goes away after the callback returns. *XtNcancelCallback* passes a pointer to an *FSBCallbackRec* as call data. Applications rarely need to specify *XtNcancelCallback*.

XtNcreateSamplerCallback To create the font sampler itself rather than letting the font selection panel create it an application must provide an *XtNcreateSamplerCallback*. *XtNcreateSamplerCallback* passes a pointer to an *FSBCreateSamplerCallbackRec* as call data. The application must fill in the *sampler* field with the widget ID of the *FontSampler* widget, and must fill in the *sampler_shell* field with the widget ID of the shell widget that contains the

sampler. An application can use this callback procedure to enclose the font sampler in another widget—for example, to display an application icon with the sampler. It can also use this procedure if it has subclassed the font sampler.

XtNfaceSelectCallback After the user chooses a new font family this callback procedure is used to pick the face selection initially provided for the new family. *XtNfaceSelectCallback* passes a pointer to an *FSBFaceSelectCallbackRec* as call data. If, after this callback has been invoked, the *new_face* field is *NULL* or is not in *available_faces*, the font selection panel chooses a face using these rules:

1. If the new family has a face with the same name as the current face, select it.

2. If not, consider similar attributes in the face name, such as Roman-Medium-Regular-Book and Italic-Oblique-Slanted. If a face that is similar to the current face is found, select it.

3. If not, select a face with one of these names, in order: Roman, Medium, Book, Regular, Light, Demi, Semibold.

4. If no matching name is found, select the first face.

Application developers typically specify an *XtNfaceSelectCallback* only if they believe they can perform better face matching than the font selection panel, or if they want to provide a face selection that is entirely different from the panel's selection.

XtNokCallback Indicates that the user wants to choose the selected options. The font selection panel disappears after the callback returns. *XtNokCallback* passes a pointer to an *FSBCallbackRec* as call data. Applications typically use the same procedure for the *XtNokCallback* as for the *XtNapplyCallback*.

XtNresetCallback The font selection panel reverts to the values last set with *XtNfontName*, *XtNfontFamily*, *XtNfontFace*, and *XtNfontSize*. The font selection panel remains. *XtNresetCallback* passes a pointer to an *FSBCallbackRec* as call data. Applications rarely need to specify an *XtNresetCallback*.

XtNvalidateCallback If an application needs to validate a font selection before accepting it, the application should provide an *XtNvalidateCallback*. The font selection panel calls *XtNvalidateCallback* before calling *XtNokCallback* or *XtNapplyCallback*. If *doit* is *False* after the call to *XtNvalidateCallback*, the *OK* or *Apply* action is canceled. *XtNvalidateCallback* passes a pointer to an *FSBValidateCallbackRec* structure as call data.

Typical uses for *XtNvalidateCallback* include verifying that exactly one font and size are selected or that an AFM file is available for the selected font. If an application rejects a selection (by setting *doit* to *False*) it should display a message that explains why the selection is rejected.

7.5.1 Callback Information

This section provides the definitions for structures that are used by the callback procedures described in the previous section.

FSBCallbackRec

```
typedef struct {
 FSBCallbackReason reason;
 String family;
 String face;
 float size;
 String name;
 String afm_filename;
 FSBSelectionType family_selection;
 FSBSelectionType face_selection;
 FSBSelectionType size_selection;
 FSBSelectionType name_selection;
 Boolean afm_present;
} FSBCallbackRec;
```

XtNokCallback, *XtNapplyCallback*, *XtNresetCallback* and *XtNcancelCallback* (see section 7.5, “Callback Procedures”) pass a pointer to an *FSBCallbackRec* structure as call data.

FSBCallbackReason is one of *FSBOK, FSBApply, FSBReset*, or *FSBCancel*.

afm_filename is assigned a value only if the *XtNgetAFM* resource is *True*.

The *…_selection* fields contain *FSBNone* if the user has made no selection, *FSBOne* if the user has made one selection, or *FSBMultiple* if the user has made multiple selections. Multiple selections are possible only if the application has set the corresponding *…Multiple* resource and the user has not modified the selection of that type of information. If the *…_selection* field is *FSBNone* or *FSBMultiple*, the corresponding data field is *NULL* or *0.0*.

afm_present is *True* if *afm_filename* is not *NULL*, and *False* if *afm_filename* is *NULL*.

FSBValidateCallbackRec

```
typedef struct {
 FSBCallbackReason reason;
 String family;
 String face;
 float size;
 String name;
 String afm_filename;
 FSBSelectionType family_selection;
 FSBSelectionType face_selection;
 FSBSelectionType size_selection;
 FSBSelectionType name_selection;
 Boolean afm_present;
 Boolean doit;
} FSBValidateCallbackRec;
```

XtNvalidateCallback passes a pointer to an *FSBValidateCallbackRec* as call data.

All fields in this structure are the same as in *FSBCallbackRec*. The *doit* field is initially *True*.

FSBFaceSelectCallbackRec

```
typedef struct {
 String *available_faces;
 int num_available_faces;
 String current_face;
 String new_face;
} FSBFaceSelectCallbackRec;
```

XtNfaceSelectCallback passes a pointer to an *FSBFaceSelectCallbackRec* as call data.

available_faces is a list of faces available in the newly selected family.

num_available_faces is the length of the *available_faces* list.

current_face is the currently selected face. If this face is one of the available faces, the pointer in *current_face* has the same value as the pointer in the *available_faces* list. Comparing the pointers for equality has the same result as comparing the pointed-to strings.

The callback should fill the *new_face* field with one of the entries in the *available_faces* field.

FSBCreateSamplerCallbackRec

```
typedef struct {
 Widget sampler;
 Widget sampler_shell;
} FSBCreateSamplerCallbackRec;
```

XtNcreateSamplerCallback passes a pointer to an *FSBCreateSamplerCallbackRec* as call data.

7.6 Procedures

This section documents the procedures supplied by the font selection panel. For all the procedures, the *widget* parameter must be a *FontSelectionBox* widget or subclass of a *FontSelectionBox* widget.

FSBDownloadFontName

```
Boolean FSBDownloadFontName (w, font_name)
 Widget w;
 String font_name;
```

FSBDownloadFontName attempts to download the font specified by *font_name*, using the specified font selection panel's resources to find the font file and to decide whether to load the font into shared VM.

FSBFindAFM

```
String FSBFindAFM (w, font_name)
 Widget w;
 String font_name;
```

FSBFindAFM returns the name of the AFM file for the specified font name, using the specified font selection panel's resources to determine where to look for the file. If no AFM file is found, *FSBFindAFM* returns *NULL*.

FSBFindFontFile

```
String FSBFindFontFile (w, font_name)
 Widget w;
 String font_name;
```

FSBFindFontFile returns the name of the font file for the specified font name, using the specified font selection panel's resources to determine where to look for the file. If no font file is found, **FSBFindFontFile** returns *NULL*.

FSBFontFamilyFaceToName

```
void FSBFontFamilyFaceToName (w, family, face,
    font_name_return)
 Widget w;
 String family;
 String face;
 String *font_name_return;
```

FSBFontFamilyFaceToName returns the font name for *family* and *face*. If *family* and *face* are not known to the font selection panel, *font_name_return* is set to *NULL*.

FSBFontNameToFamilyFace

```
void FSBFontNameToFamilyFace (w, font_name, family_return,
    face_return)
 Widget w;
 String font_name;
 String *family_return;
 String *face_return;
```

FSBFontNameToFamilyFace returns the family and face for *font_name*. If *font_name* is not known to the font selection panel, *family_return* and *face_return* are set to *NULL*.

FSBGetFaceList

```
void FSBGetFaceList (w, family, count_return, face_return,
    font_return)
 Widget w;
 String family;
 int *count_return;
 String **face_return;
 String **font_return;
```

FSBGetFaceList returns a list of the faces and a list of associated font names for the fonts specified by *family*. When the lists are no longer needed, the caller should free them with **XtFree**. The caller should not free the entries in the lists.

FSBGetFamilyList

```
void FSBGetFamilyList (w, count_return, family_return)
 Widget w,
 int *count_return;
 String **family_return;
```

FSBGetFamilyList returns a list of the font families known to the font selection panel. When the list is no longer needed, the caller should free it with **XtFree**. The caller should not free the entries in the list.

FSBGetTextDimensions

```
void FSBGetTextDimensions (w, text, font, size, x, y, dx,
    dy, left, right, top, bottom)
 Widget w;
 String text, font;
 double size, x, y;
 float *dx, *dy, *left, *right, *top, *bottom;
```

FSBGetTextDimensions returns information about the size of the text string. It can be used to avoid a potential **limitcheck** error that could result from executing **charpath** on a string.

dx and *dy* return the change in the current point that would result from showing the text at the *x, y* position, using the font at the given size. They are equivalent to the ones that **stringwidth** returns.

left, *right*, *top*, and *bottom* return the bounding box of the imaged text. They are the ones that would result from the following code:

```
(text) false charpath flattenpath pathbbox
```

There is no danger of a **limitcheck** error if the resulting path exceeds the maximum allowed path length.

FSBMatchFontFace

```
Boolean FSBMatchFontFace (w, old_face, new_family,
    new_face_return)
 Widget w;
 String old_face;
 String new_family;
 String *new_face_return;
```

FSBMatchFontFace attempts to find a face in *new_family* that is similar to *old_face*. It uses the same rules as the default *XtNfaceSelectCallback*, with the following results:

- If the font selection panel does not know *new_family*, *new_face_return* is *NULL* and **FSBMatchFontFace** returns *False*.
- If the font selection panel succeeds in finding a close match, it returns the new face in *new_face_return* and returns *True.*
- If the font selection panel cannot find a close match, it stores the closest it can find (a “regular” face or, failing that, the first face) in *new_face_return* and returns *False*.

FSBRefreshFontList

```
void FSBRefreshFontList (w)
 Widget w;
```

FSBRefreshFontList instructs the font selection panel to refresh its font lists. An application should call this procedure only when new fonts have been installed.

FSBSetFontFamilyFace

```
void FSBSetFontFamilyFace (w, font_family, font_face,
    font_family_multiple, font_face_multiple)
 Widget w;
 String font_family;
 String font_face;
 Bool font_family_multiple;
 Bool font_face_multiple;
```

Calling **FSBSetFontFamilyFace** is equivalent to calling **XtSetValues** with the *XtNuseFontName, XtNfontFamily, XtNfontFace, XtNfontFamilyMultiple*, and *XtNfontFaceMultiple* resources. *XtNuseFontName* is set to *False.*

FSBSetFontName

```
void FSBSetFontName (w, font_name, font_name_multiple)
 Widget w;
 String font_name;
 Bool font_name_multiple;
```

Calling **FSBSetFontName** is equivalent to calling **XtSetValues** with the *XtNuseFontName*, *XtNfontName*, and *XtNfontNameMultiple* resources. *XtNuseFontName* is set to *True*.

FSBSetFontSize

```
void FSBSetFontSize (w, font_size, font_size_multiple)
 Widget w;
 double font_size;
 Bool font_size_multiple;
```

Calling **FSBSetFontSize** is equivalent to calling **XtSetValues** with the *XtNfontSize* and *XtNfontSizeMultiple* resources.

FSBUndefineUnusedFonts

```
void FSBUndefineUnusedFonts (w)
 Widget w;
```

FSBUndefineUnusedFonts undefines all fonts that were downloaded for previewing but are not the currently previewed font. Since this happens automatically when the user activates the OK, Apply, Reset, or Cancel button,

FSBUndefineUnusedFonts should be called only if the application has popped the font selection panel down without waiting for the user to activate the OK or Cancel button.

8 The Motif Font Sampler

The Motif font sampler can be popped up from the font selection panel to view multiple fonts at the same time. This section provides information about:

- Using the font sampler
- Resources (listing and description)
- Callbacks and procedures

By default, the font selection panel creates the font sampler and pops it up and down. The application can intervene using the *XtNcreateSamplerCallback* procedure of the font panel.

The header file is <*DPS/FontSample.h*>.

The class pointer is *fontSamplerWidgetClass*.

The class name is *FontSampler*.

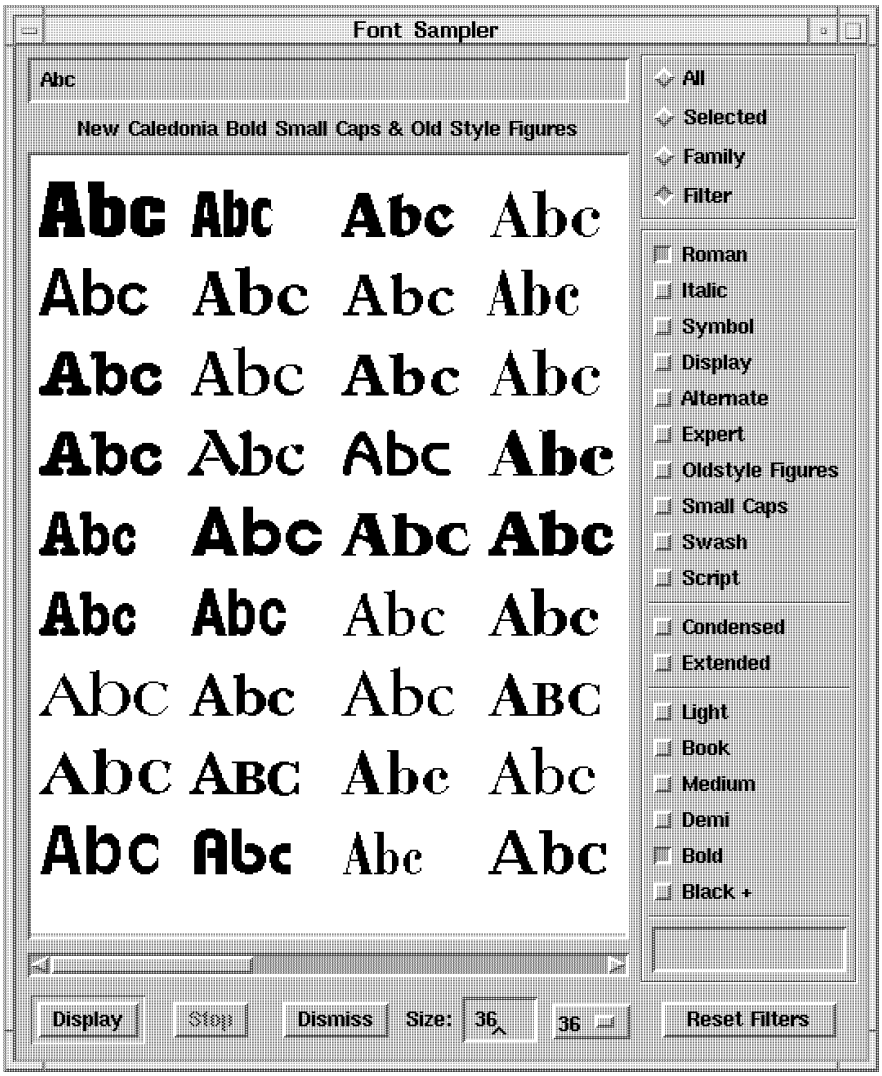
The *FontSampler* widget is a subclass of *XmManager*.

8.1 Using the Motif Font Sampler

The font sampler allows the user to view multiple fonts at the same time and to choose among them. It provides a set of filters that narrow the choices to fonts with particular characteristics. Figure 2 shows a font sampler displaying the letters “Abc” using selected filters.

At the top of the font sampler, a text field allows the user to specify the text to be previewed. Below the type-in region is a display area on the left and a selection area containing boxes with toggles on the right. The top box allows the user to select a display criterion; below that are the choices for filtering fonts.

Figure 2 *The font sampler*



To display a set of fonts the user:

1. Types some text into the text field.

2. Chooses a display criterion and one or more font selection criteria, as described below.

3. Activates the Display button.

If the user activates the Display button during an ongoing display, the font sampler restarts the display. If the user activates the Stop button, any ongoing display is stopped.

To remove the font sampler from the screen, activate the Dismiss button.

8.1.1 Display Criteria

When the user activates the Display button, the font sampler begins to show fonts, using the current display criteria:

- If the All toggle is set, all fonts are displayed.
- If the Selected toggle is set, the font selected in the associated font selection panel is displayed.
- If the Family toggle is set, all fonts in the family selected in the associated font selection panel are displayed.
- If the Filter toggle is set, fonts that match the current set of filters are displayed.

In all cases, the size field and menu control the size of the displayed font.

8.1.2 Font Selection Criteria

The filter check boxes determine which fonts are displayed when the Filter toggle is set. There are four sets of filters:

- The first set describes general classes of fonts: Roman, Italic, Symbol, Small Caps, Script, and so on. (Note that “Roman” in this context means not Italic and not Symbol).
- The second set describes condensed or expanded fonts.
- The third set describes font weights: Light, Medium, Bold, and so on.
- The fourth set is a filter text field that is used for general matching.

A font will be displayed only if it matches one of the check boxes in each set that has any boxes checked. If the filter text field is not empty, the font name must also contain the string in that text field. (For the mathematically minded, the check boxes form a conjunction of disjunctions.)

Consider the following two examples:

- In the first set, the Italic box is checked. In the second set, no boxes are checked. In the third set, the Bold and Demi boxes are checked. The text field is empty. This combination matches any italic font that is either bold or demi. Note that the font sampler does not compare fonts against the filters in the second set, because no boxes are checked in that set.
- In the first set, Roman and Italic are checked. In the second set, Condensed is checked. In the third set, no boxes are checked. The text field contains the string “Garamond”. This combination matches all nonsymbol, condensed fonts that have the string “Garamond” in their name.

The filtering process is based on searching for strings in the font's full name. Each check box has a set of strings that it matches. The Italic check box, for example, matches the strings "Italic", "Oblique", "Slanted", or "Kursiv".

Note: Font naming does not follow a simple set of rules, so the results of a match might be unexpected. For example, if the user selected Symbol, any font that contained the string "Symbol" in its name would be matched, even if the font did not actually contain symbols. Similarly, a font that was "Ultra Condensed" would match the Black+ check box because "Ultra" is one of the strings that matches the Black+ category. (While some fonts use the phrase "Ultra Condensed" to mean very condensed, others use it to mean ultra-heavy and condensed.)

Any changes to the currently displayed text, the size of the sampled fonts, the check boxes, or the filter text take effect immediately. However, changing the display criteria (All, Selected, Family, and Filtered) does not affect the current display until the user clicks the Display button.

Selecting a filter automatically changes the display type to Filtered.

If the user activates the Reset Filters button, all check boxes are toggled to *off* and the text filter is reset to be empty.

Clicking any mouse button on a displayed font sample displays the font name above the work area. The font becomes selected in the associated font selection panel. The size selected in the font selection panel is not affected by the size selected in the sampler.

8.2 Motif Font Sampler Resources

The following section describes the font sampler resources. The resource set table is followed by a brief description of each resource.

Table 12 *Motif font sampler resource set*

Name	*Class*	*Default*	*Type*	*Access*
XtNfontSelectionBox	XmCFontSelectionBox	*NULL*	XtRWidget	C
XtNminimumHeight	XmCMinimumHeight	100	XtRDimension	CSG
XtNminimumWidth	XmCMinimumWidth	100	XtRDimension	CSG
XtNnoFamilyFontMessage	XmCNoFamilyFontMessage	*See description*	XmRXmString	CSG
XtNnoFontMessage	XmCNoFontMessage	"There are no fonts!"	XmRXmString	CSG
XtNnoMatchMessage	XmCNoMatchMessage	"No fonts match filters"	XmRXmString	CSG
XtNnoRoomMessage	XmCNoRoomMessage	*See description*	XmRXmString	CSG
XtNnoSelectedFamilyMessage	XmCNoSelectedFamilyMessage	*See description*	XmRXmString	CSG

Table 12 *Motif font sampler resource set (Continued)*

Name	*Class*	*Default*	*Type*	*Access*
XtNnoSelectedFontMessage	XmCNoSelectedFontMessage	*See description*	XmRXmString	CSG
XtNsizeCount	XmCSizeCount	10	XtRInt	CSG
XtNsizes	XmCSizes	*See description*	XtRFloatList	CSG

8.2.1 Resource Descriptions

XtNfontSelectionBox Specifies the *FontSelectionBox* widget associated with a *FontSampler* widget. This resource must be specified when the font sampler is created, and cannot be changed.

XtNminimumHeight Specifies the minimum height for the work area. If the user resizes the font sampler and the work area becomes shorter than *XtNminimumHeight*, a vertical scroll bar appears and allows scrolling. Default is 100.

XtNminimumWidth Specifies the minimum width for the work area. If the user resizes the font sampler and the work area becomes narrower than *XtNminimumWidth*, a horizontal scroll bar appears and allows scrolling. Default is 100.

XtNnoFamilyFontMessage Specifies the compound string the font sampler displays if the selected font family has no fonts. This should not happen. Default is "Selected family has no fonts!"

XtNnoFontMessage Specifies the compound string the font sampler string displays if there are no fonts to be shown. This should not happen. Default is "There are no fonts!"

XtNnoMatchMessage Specifies the compound string the font sampler displays if no fonts match the selected filters. Default is "No fonts match filters."

XtNnoRoomMessage Specifies the compound string the font sampler displays if the work area is too small to show a single font sample. Default is "Current size is too large or panel is too small."

XtNnoSelectedFamilyMessage Specifies the compound string the font sampler displays if no font family is selected but the user chooses to display the selected family. Default is "No family is currently selected."

XtNnoSelectMessage Specifies the compound string the font sampler displays if no font is selected but the user chooses to display the selected font. Default is "No font is currently selected."

XtNsizeCount Specifies the number of entries in the *XtNsizes* resource. Default is 10.

XtNsizes Specifies the list of sizes to present in the menu. Default is 8, 10, 12, 14, 16, 18, 24, 36, 48, 72.

8.2.2 Children of the Motif Font Sampler

The following resources provide access to the descendants of the font selection panel. These resources cannot be changed. All are of type *XtRWidget*.

Table 13 *Motif Font sampler child resource set*

Name	*Class*	*Type*	*Access*
XtNallToggleChild	XtCReadOnly	XtRWidget	G
XtNareaChild	XtCReadOnly	XtRWidget	G
XtNclearButtonChild	XtCReadOnly	XtRWidget	G
XtNdismissButtonChild	XtCReadOnly	XtRWidget	G
XtNdisplayButtonChild	XtCReadOnly	XtRWidget	G
XtNfilterBoxChild	XtCReadOnly	XtRWidget	G
XtNfilterFrameChild	XtCReadOnly	XtRWidget	G
XtNfilterTextChild	XtCReadOnly	XtRWidget	G
XtNfilterToggleChild	XtCReadOnly	XtRWidget	G
XtNpanelChild	XtCReadOnly	XtRWidget	G
XtNradioBoxChild	XtCReadOnly	XtRWidget	G
XtNradioFrameChild	XtCReadOnly	XtRWidget	G
XtNscrolledWindowChild	XtCReadOnly	XtRWidget	G
XtNselectedFamilyToggleChild	XtCReadOnly	XtRWidget	G
XtNselectedToggleChild	XtCReadOnly	XtRWidget	G
XtNsizeLabelChild	XtCReadOnly	XtRWidget	G
XtNsizeOptionMenuChild	XtCReadOnly	XtRWidget	G
XtNsizeTextFieldChild	XtCReadOnly	XtRWidget	G
XtNstopButtonChild	XtCReadOnly	XtRWidget	G
XtNtextChild	XtCReadOnly	XtRWidget	G

8.3 Callbacks

XtNdismissCallback *XtNdismissCallback* indicates that the user has dismissed the font sampler. The call data is *NULL*.

8.4 Procedures

FSBCancelSampler

```
void FSBCancelSampler (w)
 Widget w;
```

FSBCancelSampler cancels any display currently in progress. It can be used if the creator of the font sampler disables the sampler. If the user or application pops down the font selection panel, **FSBCancelSampler** does not have to be called; the font selection panel calls the appropriate procedures itself.

Appendix A

Locating PostScript Language Resources

Applications that use the PostScript language need to locate files that describe and contain PostScript language objects. The files may be Adobe Font Metric (AFM) files, font outline files, PostScript language procedure sets, forms, patterns, encodings, or any named PostScript language object. They are collectively referred to as PostScript language resource files, or *resource files*.

In many cases, resource files are installed system-wide—for example, the AFM files for the fonts that reside on a system's PostScript printers. In other cases, resource files are private to a user—for example, private font outlines that a user has purchased or procedure sets for a private application. PostScript language resource database files, or *resource database files*, allow applications to locate resource files uniformly.

This appendix contains information about locating resources, including:

- The structure of resource database files.
- The predefined resource type names.
- Facilities for locating resource database files.
- Procedures and type definitions for locating resources.
- Memory management and error handling.

Appendix B describes the *makepsres* utility, which you can use to create resource database files.

TK

A.1 Resource Database Files

This section describes resource database files, which can be used to locate resource files, including:

- Description of the format
- Information about the different sections
- A sample resource database file

A.1.1 Format of a Resource Database File

The following restrictions and requirements exist for the format of a resource database file:

- No line may exceed 255 characters plus the line termination character.
- A backslash (\) quotes any character. For example, the sequence \ABC represents the characters ABC. In most sections of the file, you may continue any line by ending it with a backslash immediately before the newline character (see section A.1.2, “Components of a Resource Database File”).
- A section terminator begins with a period. If you begin any other line with a period, you must precede the period with a backslash.
- All lines in the file are case-sensitive.
- To include comments on any line, precede them with a percent sign. To avoid making a percent sign a comment, precede it with a backslash.
- Trailing blanks and tab characters are ignored everywhere in the file, but they do count toward the 255-character line length limit.

A.1.2 Components of a Resource Database File

A resource database file consists of several components, which must appear in the database in the following order:

- An identifying string (required)
- A list of resource types in this resource database file (required)
- A directory path (optional)
- The data for each resource type (required)

Identifying String Component

The first line of a resource database file must contain either the constant string *PS-Resources-1.0* or the constant string *PSResources-Exclusive-1.0*. The difference between the two is explained in section A.3, “Locating Resource Database Files.”

Resource Types Component

The resource types component lists the resource types described by the file. Each line is the name of a single resource type, terminated by a newline character. The resource types component is terminated by a line containing a single period. Any string can be used to identify a resource type; the predefined resource types are defined in section A.2.

Directory Component

The directory component is an optional single line that identifies the directory prefix to be added to all file names in the resource database file. The component consists of a slash (/) followed by the directory prefix. (In operating systems where a slash is the first character of a fully specified path, the line must begin with two slashes.) If the directory component is not present, the directory prefix becomes, by default, the directory containing the resource database file.

Resource Data Components

Each resource type requires a data component. The data components must be presented in the same order as the corresponding identifiers in the resource types component.

Each data component consists of a single line identifying the resource type, followed by lines of resource data for that type, followed by a line containing a single period.

Each line of resource data contains:

- The name of the resource. If the name contains an equal sign, precede the equal sign with a backslash.
- A single or double equal sign (= or = =).
- The name of the file that contains the resource. The file name may be an absolute or relative path name. If relative, it is interpreted relative to the directory prefix as specified above in the directory component description. However, a double equal sign forces the file name to be interpreted as absolute. In that case, the prefix is not used.

For some special predefined resource types the file name is replaced by some other kind of data; see section A.2. In these cases the directory prefix does not apply. (It is as if every line of resource data were specified with a double equal sign.)

A.1.3 Resource Database File Example

This is a sample resource database file for fonts in the Trajan family. The next section describes the resource types used in this example.

Example A.1 *Resource database file for fonts in the Trajan family*

```
PS-Resources-1.0
FontOutline      % This section lists resource types
FontPrebuilt
FontAFM
FontFamily
FontBDF
FontBDFSizes
.                % This line ends resource type listing
//usr/local/PS/resources
FontOutline      % This section lists font outline files
Trajan-Bold=Trajan-Bold
Trajan-Regular=Trajan-Regular
.
FontPrebuilt
Trajan-Bold=Trajan-Bold.bepf
Trajan-Regular=Trajan-Regular.bepf
.
FontAFM
Trajan-Bold=Trajan-Bold.afm
Trajan-Regular=Trajan-Regular.afm
.
FontFamily
Trajan=Bold,Trajan-Bold,Regular,Trajan-Regular
.
FontBDF
Trajan-Regular18-75-75=Trajan-Regular.18.bdf
Trajan-Regular24-75-75=Trajan-Regular.24.bdf
Trajan-Regular36-75-75=Trajan-Regular.36.bdf
Trajan-Regular48-75-75=Trajan-Regular.48.bdf
Trajan-Bold18-75-75=Trajan-Bold.18.bdf
Trajan-Bold24-75-75=Trajan-Bold.24.bdf
Trajan-Bold36-75-75=Trajan-Bold.36.bdf
Trajan-Bold48-75-75=Trajan-Bold.48.bdf
.
FontBDFSizes
Trajan-Regular=18-75-75,24-75-75,36-75-75,48-75-75
Trajan-Bold=18-75-75,24-75-75,36-75-75,48-75-75
```

A.2 Predefined Resource Types

The following table lists the name and contents of the predefined resource types. Each resource line contains the name of the resource, a single equal sign, and the name of the file containing the resource or other relevant information, as described in the table. Examples for several of the resource types can be found in section A.1.3, “Resource Database File Example.”

Table A.1 *Resource types*

Resource	*Contents*
FontOutline	The file contains PostScript language character outline programs.
FontPrebuilt	The file contains a set of prebuilt font bitmaps. Currently, only the Display PostScript system uses this format.
FontAFM	The file is an AFM file.
FontBDF	The file contains bitmap font data in Bitmap Distribution Format (BDF).

FontBDFSizes	If the resource type is *FontBDFSizes*, the file name in the resource line is replaced by a list of the *FontBDF* resources in the current file. Each entry consists of the point size, the *x* resolution, and the *y* resolution of the BDF file, separated by a single hyphen (–) character. The entries are separated with a single comma (,) character. Each entry may be appended to the resource name on the line to yield a valid *FontBDF* resource. The directory prefix does not apply to this resource type.
	
	In the sample file in A.1.3, “Resource Database File Example,” the line
	
	`Trajan-Regular=18-75-75,24-75-75,36-75-75,48-75-75`
	
	indicates that the Trajan-Regular font has four BDF files available, at 18, 24, 36 and 48 points. All files are at 75 dots per inch in x and y. The name of each FontBDF resource is formed by concatenating Trajan-Regular with one of the size specifications, yielding, for example,
	
	`Trajan-Regular18-75-75`

FontFamily	If the resource type is *FontFamily*, the file name in the resource line is replaced by a list of *FontOutline* resource names in the current file that belong to this font family. Each resource name is preceded by the face name for the font. The names are separated by a single comma (,) character; use a backslash to quote a comma within a font name. The directory prefix does not apply to this resource type. In the sample file in section A.1.3, “Resource Database File Example,” the line `Trajan=Bold,Trajan-Bold,Regular,Trajan-Regular` indicates that the Trajan family contains two faces: Bold, with the font name Trajan-Bold, and Regular, with the font name Trajan-Regular. The correspondence between face names and font names is not always as straightforward as in this example.

Table A.1 *Resource types (Continued)*

Resource	*Contents*
Form	The file contains a **Form** definition; see section 3.9.2 of *PostScript Language Reference Manual, Second Edition.*
Pattern	The file contains a **Pattern** definition; see section 3.9.2 of *PostScript Language Reference Manual, Second Edition.*
Encoding	The file contains a character set encoding; see section 3.9.2 of *PostScript Language Reference Manual, Second Edition.*
ProcSet	The file contains a named set of PostScript language procedures implementing some piece of an application's prolog.
mkpsresPrivate	The *makepsres* utility generates and manipulates resource database files. This section contains private information stored by *makepsres* to help it in future invocations. For more information about *makepsres*, consult Appendix B.

Further predefined types will be added to represent additional resources as needed.

A.3 Locating Resource Database Files

A user's *PSRESOURCEPATH* environment variable consists of a list of directories separated by colons. (Systems without environment variables must use an alternate way of expressing the user's preference.) Procedures that look for resource database files search each directory named in the *PSRESOURCEPATH* environment variable.

Each component (for example, the font selection panel and the TranScript™ software package) has a default place where it looks for resource files. The default places may be different for each component and are determined in a component-dependent way, usually at system build time.

Two adjacent colons in a *PSRESOURCEPATH* path represent the list of default places in which a component looks for PostScript language resources.

The *PSRESOURCEPATH* variable defaults to "::" if no value is specified. This is the normal value for users who have not installed private resources. Users with private resources should end the path with a double colon if they want their resources to override the system defaults, or begin it with a double colon if they don't want to override system defaults. Colons in the path can be quoted in a system-dependent way; on UNIX® systems, a backslash quotes colons.

A typical *PSRESOURCEPATH* is the following:

```
::/proj/ourproj/PS:/user/smith/ps
```

The sample path above instructs procedures that locate resource database files to first look in the default place, wherever it may be, then to search the directory */proj/ourproj/PS*, and then search the directory */user/smith/ps*. The user does not need to know the location of the default resource database files.

On UNIX systems, resource database files end with the suffix *.upr* (for UNIX PostScript resources). The principal resource database file in a directory is named *PSres.upr.*

- If the first line of a *PSres.upr* file is *PS-Resources-Exclusive-1.0*, the *PSres.upr* file is the only resource database file in its directory.

- If the first line of a *PSres.upr* file is *PS-Resources-1.0*, or if there is no *PSres.upr* file, any file in the same directory with the suffix *.upr* is a resource database file. For example, the sample file shown in A.1.3, “Resource Database File Example,” might be called *Trajan.upr.*

If a *PSres.upr* file begins with *PS-Resources-Exclusive-1.0*, the resource location procedures run more quickly since they don’t need to look for other *.upr* files. However, users will then have to update *PSres.upr* whenever new resources are installed.

A.4 Type Definitions and Procedures for Resource Location

If you are writing an application or a library that needs to locate PostScript language resource files, use the resource location library *libpsres.a*. This library contains procedures that locate and parse resource database files and return lists of resource files. The header file for *libpsres.a* is <*DPS/PSres.h*>.

Resource location procedures represent resource types as character strings. This allows matching of arbitrary strings in the resource type list of a resource database file. Several variables are available for matching:

```
extern char *PSResFontOutline, *PSResFontPrebuilt,
    *PSResFontAFM,*PSResFontBDF, *PSResFontFamily,
    *PSResFontBDFSizes,*PSResForm, *PSResPattern,
    *PSResEncoding, *PSResProcSet;
```

The variables evaluate to the appropriate character string; for example, the value of *PSResFontOutline* is “FontOutline”. Using the variables instead of the strings themselves allows the compiler to find spelling errors within your application that would otherwise go undetected.

In the following procedure definitions, the phrase *resource location procedure* refers to **ListPSResourceFiles**, **EnumeratePSResourceFiles**, or **ListPSResourceTypes**, but not to **CheckPSResourceTime**.

A.4.1 Type Definitions

PSResourceEnumerator

```
typedef int *(PSResourceEnumerator)(/*
 char *resourceType,
 char *resourceName,
 char *resourceFile,
 char *private*/);
```

A *PSResourceEnumerator* procedure is used with **EnumeratePSResourceFiles**.

PSResourceSavePolicy

```
typedef enum {
 PSSaveReturnValues,
 PSSaveByType,
 PSSaveEverything
} PSResourceSavePolicy;
```

PSResourceSavePolicy enumerates the save policies used by **SetPSResourcePolicy**.

A.4.2 Procedures

CheckPSResourceTime

```
int CheckPSResourceTime (psResourcePathOverride,
                         defaultPath)
 char *psResourcePathOverride;
 char *defaultPath;
```

CheckPSResourceTime checks whether the access times of directories in a path have changed since the directories were read in.

psResourcePathOverride provides a path that overrides the environment resource path. On UNIX systems, it replaces the *PSRESOURCEPATH* environment variable. The value is usually *NULL*. To quote colons in the path, use a backslash.

defaultPath is the path that is inserted between adjacent colons in the resource path. It may be *NULL*.

- If either path value differs from that used in the previous call to any procedure in this library, **CheckPSResourceTime** returns 1.

- If neither path has changed since the previous call to the library, **CheckPSResourceTime** determines whether the modification time for any directory described in the paths is more recent than the latest modification time when the directories were scanned for resource files and, if so, returns 1. Otherwise **CheckPSResourceTime** returns 0.

CheckPSResourceTime does not free storage and cannot make invalid storage that was previously returned by **ListPSResourceFiles** or **ListPSResourceTypes**.

If **CheckPSResourceTime** returns 1, the caller can then call

```
FreePSResourceStorage(1)
```

This forces future calls to resource location procedures to reload all resource databases.

EnumeratePSResourceFiles

```
void EnumeratePSResourceFiles (psResourcePathOverride,
    defaultPath, resourceType, resourceName,
    enumerator, private)
 char *psResourcePathOverride;
 char *defaultPath;
 char *resourceType;
 char *resourceName;
 PSResourceEnumerator enumerator;
 char *private;
```

EnumeratePSResourceFiles lists PostScript language files giving applications complete control over saving file names. Applications that do not need this level of control should use **ListPSResourceFiles** instead.

EnumeratePSResourceFiles calls the procedure specified by *enumerator* for each resource that matches the *resourceType* and (if non-*NULL*) *resourceName*. The enumerator procedure has to copy the resource name and resource file into nonvolatile storage before returning. The *resourceType* parameter is passed to the enumerator for information only; it does not have to be copied. If the enumerator procedure returns a nonzero value, **EnumeratePSResourceFiles** returns without enumerating further resources.

EnumeratePSResourceFiles causes minimal state to be saved—for example, which resource files contain which types of resources. To free the saved state, call

```
FreePSResourceStorage(1)
```

psResourcePathOverride provides a path that overrides the environment resource path. On UNIX systems, it replaces the *PSRESOURCEPATH* environment variable. The value is usually *NULL*. To quote colons in the path, use a backslash.

TK

defaultPath is the path inserted between adjacent colons in the resource path. The value may be *NULL*.

resourceType indicates the type of resource desired.

resourceName indicates the requested resource name. If the name is *NULL*, the procedure returns a list of all resource names for the type in *resourceType*.

enumerator provides a procedure that is called for each resource name.

private specifies data to be passed uninterpreted to the enumerator.

FreePSResourceStorage

```
void FreePSResourceStorage (everything)
 int everything;
```

The subroutine library normally keeps internal state to avoid reading directory files each time. Calling **FreePSResourceStorage** frees any storage currently used.

- If *everything* is nonzero **FreePSResourceStorage** completely resets its state. No information is retained.
- If *everything* is zero, **FreePSResourceStorage** allows the library to keep minimal information, normally about which files in the search path contain which resource types.

Calling a resource location procedure with a different value of *psResourcePathOverride* or of *defaultPath* from the previous call implicitly makes the call

```
FreePSResourceStorage(1)
```

FreePSResourceStorage invalidates any string pointers returned by previous calls to **ListPSResourceFiles** or **ListPSResourceTypes**.

ListPSResourceFiles

```
int ListPSResourceFiles (psResourcePathOverride,
    defaultPath, resourceType, resourceName,
    resourceNamesReturn, resourceFilesReturn)
 char *psResourcePathOverride;
 char *defaultPath;
 char *resourceType;
 char *resourceName;
 char **resourceNamesReturn;
 char **resourceFilesReturn;
```

ListPSResourceFiles lists PostScript language resource files.

psResourcePathOverride provides a path that overrides the environment resource path. On UNIX systems, it replaces the *PSRESOURCEPATH* environment variable. The value is usually *NULL*. To quote a colon in the path, use a backslash.

defaultPath is the path that is inserted between adjacent colons in the resource path. *defaultPath* may be *NULL*.

resourceType indicates the type of resource desired.

resourceName indicates the desired resource name. If *resourceName* is *NULL*, **ListPSResourceFiles** returns a list of all resource names of type *resourceType*.

resourceNamesReturn returns a list of the resource names.

resourceFilesReturn returns a list of the resource file names as absolute path names. Backslash quotes are removed from all strings, with the exception of backslashes that precede commas in the file name. This supports comma quoting for the *FontFamily* resource type.

The *resourceNamesReturn* and *resourceFilesReturn* arrays always have the same number of entries, equal to the return value.

The *resourceNamesReturn* and *resourceFilesReturn* arrays should be freed with **PSResFree** when they are no longer needed. The individual strings should not be freed. They remain valid until a resource location procedure is called with a different value of *psResourcePathOverride* or *defaultPath*, or until **FreePSResourceStorage** is called.

If a particular resource name occurs more than once in the same or in different resource directories, all occurrences are returned, in the following order:

- All resources for a particular directory entry in the resource search path occur before any entries for a later directory.
- For a particular directory, all resources found in *PSres.upr* files occur before any entries found in subsidiary resource directory files.

ListPSResourceFiles returns the number of entries in the *resourceNamesReturn* array. A return value of 0 means that no resources meeting the specification could be found. In that case, *resourceNamesReturn* and *resourceFilesReturn* are not modified.

Applications that need complete control over saving the file names should use **EnumeratePSResourceFiles**.

ListPSResourceTypes

```
int ListPSResourceTypes (psResourcePathOverride,
    defaultPath, resourceTypesReturn)
 char *psResourcePathOverride;
 char *defaultPath;
 char **resourceTypesReturn;
```

Applications can call **ListPSResourceTypes** to determine which resource types are available.

psResourcePathOverride provides a path that overrides the environment resource path. On UNIX systems, that path replaces the *PSRESOURCEPATH* environment variable. The value is usually *NULL*. To quote colons in the path, use a backslash.

defaultPath is the path that is inserted between adjacent colons in the resource path. The value may be *NULL*.

resourceTypesReturn returns a list of resource types.

The *resourceTypesReturn* array should be freed with **PSResFree** when it is no longer needed. The individual strings should not be freed; they remain valid until a resource location procedure is called with a different value of *psResourcePathOverride* or of *defaultPath*, or until **FreePSResourceStorage** is called with *everything* nonzero.

The returned resource types are merged to result in a nonduplicating list. The special type *mkpsresPrivate* is never returned.

ListPSResourceTypes returns the number of entries in the *resourceTypesReturn* array. A return value of 0 means that no resource types could be found. In that case, *resourceTypesReturn* is not modified.

SetPSResourcePolicy

```
void SetPSResourcePolicy (policy, willList, resourceTypes)
 PSResourceSavePolicy policy;
 int willList;
 char **resourceTypes;
```

An application can use **SetPSResourcePolicy** to provide the resource library with information about the expected pattern of future calls to **ListPSResourceFiles**.

policy determines the save policy used. It is of type *PSResourceSavePolicy* and may be one of the following:

- *PSSaveEverything*. The first time **ListPSResourceFiles** or **ListPSResourceTypes** is called with a particular set of values for *psResourcePathOverride* and *defaultPath*, it reads all resource directory files in the specified paths and caches all the information in them. Future calls will use the cache and not read the file system.

- *PSSaveByType*. **ListPSResourceFiles** saves information about the resource types in *resourceTypes*. Calls to **ListPSResourceFiles** for resource types not in the *resourceTypes* list may or may not save values. In that case, it is undefined whether and how much information is saved.

- *PSSaveReturnValues*. **ListPSResourceFiles** saves the returned strings but may save little else. Subsequent calls usually access the file system. It is undefined whether and how much other information is saved, but applications can expect it to be minimal.

You cannot completely disable saving since the saved strings are returned by **ListPSResourceFiles**.

willList is nonzero if the application expects to list resources by passing *NULL* to **ListPSResourceFiles** in the *resourceName* parameter.

resourceTypes is a *NULL*-terminated list of the resource types the application expects to use, or *NULL*.

Note that *willList, policy,* and *resourceTypes* are just hints; **ListPSResourceFiles** works correctly regardless of their values. It may, however, work more slowly.

Calling **SetPSResourcePolicy** more than once changes the future behavior of **ListPSResourceValues** but has no effect on the previously saved state.

Applications that need complete control over saving the names can use **EnumeratePSResourceFiles** instead of **ListPSResourceFiles**.

A.5 Memory Management and Error Handling

An application using the resource location library may provide its own implementation of **malloc**, **realloc**, and **free** by assigning values to the external variables *PSResMalloc*, *PSResRealloc*, and *PSResFree*.

TK

PSResMallocProc

```
typedef char *(*PSResMallocProc)(/*
  int size */);

extern PSResMallocProc PSResMalloc;
```

PSResReallocProc

```
typedef char *(*PSResReallocProc)(/*
 char *ptr,
 int size */);

extern PSResReallocProc PSResRealloc;
```

PSResFreeProc

```
typedef void (*PSResFreeProc)(/*
 char *ptr */);

extern PSResFreeProc PSResFree;
```

The procedures must provide the following additional semantics beyond that supplied by the system allocation routines:

- **PSResMalloc** and **PSResRealloc** must never return *NULL*; if they return at all they must return the storage. They must not return *NULL* even if passed a zero size.
- **PSResFree** must return if passed *NULL*, and do nothing else.
- **PSResRealloc** must allocate storage if passed a *NULL* pointer.

The default routines give an error message and terminate if they are unable to allocate the requested storage.

If the resource location library encounters a resource database file that does not conform to the standard format, a warning handler is called. The default warning handler prints a warning message with the file name on *stderr* and continues, ignoring information it cannot parse. A different warning handler can be installed by assigning a value to the external variable *PSResFileWarningHandler*.

PSResFileWarningHandlerProc

```
typedef void (*PSResFileWarningHandlerProc)
 (/* char *fileName,
     char *extraInfo*/);

extern PSResFileWarningHandlerProc
      PSResFileWarningHandler;
```

Appendix B

The makepsres Utility

The *makepsres* utility creates resource database files. Resource database files can be used to locate PostScript language resources that are used by the font selection panel and other Adobe software. If an application needs to locate PostScript language resources, it uses the facilities described in Appendix A, “Locating PostScript Language Resources.”

This appendix provides information about the *makepsres* utility in a format similar to a UNIX manual page.

To invoke *makepsres* in the default mode, type:

```
makepsres
```

Resource installation scripts should invoke *makepsres* automatically.

B.1 Overview of Functionality

The complete command line syntax for *makepsres* is:

```
makepsres [options] directory ...
```

makepsres creates a resource database file containing all the resources in all directories specified on the command line.

- If the list of directories contains “**–**”, *makepsres* reads from *stdin* and expects a list of directories separated by space, tab, or newline.
- If the list of directories is empty, it is taken to be the current directory.
- If all specified directories have a common initial prefix, *makepsres* extracts it as a directory prefix in the new resource database file.

makepsres uses existing resource database files to assist in identifying files. By default, *makepsres* creates a new resource database file containing all of the following that apply:

- Resource files found in the directories on the command line.

TK

- Resource files pointed to by the resource database files in the directories on the command line.

- Resource entries found in the input resource database files. These entries are copied if the files they specify still exist and are located in directories not specified on the command line.

makepsres uses various heuristics to identify files. A file that is of a private resource type or that does not conform to the standard format for a resource file must be specified in one of the following ways:

- Be identified by the user by running *makepsres* in interactive mode

- Have been preloaded into a PostScript resource database file used for input

- Begin with the following line:

```
%!PS-Adobe-3.0 Resource-<resource-type>
```

If you run *makepsres* in *discard mode* (using the **–d** command line option), it does not copy resource entries from the input resource database files. In that case, the output file consists only of entries from the directories on the command line. The input resource database files are only used to assist in identifying files.

If you run *makepsres* in *keep mode* (using the **–k** command line option), it includes in the output file all resource entries in the input resource database files, even entries for files that no longer exist or are located in directories specified on the command line.

B.2 Command Line Options

–o *filename* Writes the output to the specified file name. **"–o –**" writes to *stdout.* If the **–o** option is not specified, *makepsres* creates a *PSres.upr* file in the current directory and writes the output to that file.

–f *filename* Uses information from the specified file to assist in resource typing. The file must be in PostScript resource database file format (see A.1.1, "Format of a Resource Database File"). Multiple **–f** options may be specified. **"–f –**" uses *stdin* as an input file and may not be used if "**–**" is specified as a directory on the command line. It is not necessary to use **–f** for files that are in a directory on the command line.

–dir *filename* Specifies that *filename* is a directory. This option is only needed if the directory name can be confused with one of the command line options.

–d Specifies discard mode. If **–d** is used, *makepsres* does not copy resource entries from the input resource database files. In that case, the output file consists solely of entries from the directories on the command line. The input resource database files are only used to assist in identifying files.

–e Tells *makepsres* to mark the resulting *PSres.upr* file as exclusive. This option makes the resource location library run more quickly since it does not have to look for other resource database files. It becomes necessary, however, to run *makepsres* whenever new resources are added to the directory, even if the resources come with their own resource database file.

–i Specifies interactive mode. In interactive mode, the user is queried for the resource type of any file encountered by *makepsres* that it cannot identify. If **–i** is not specified, *makepsres* assumes an unidentifiable file is not a resource file.

–k Specifies keep mode. If **–k** is used, *makepsres* includes all resource entries from the input resource database files in the output. This includes entries for files that no longer exist.

–nr Specifies nonrecursive mode. *makepsres* normally acts recursively: it looks for resource files in subdirectories of any specified directory. If **–nr** is used, *makepsres* does not look in subdirectories for resource files.

–nb Does not back up the output file if it already exists.

–p Specifies no directory prefix. If **–p** is used, *makepsres* does not try to find a common directory prefix among the specified directories.

–q Ignores unidentifiable files instead of warning about them; “be quiet.”

–s Specifies strict mode. If **–s** is used, *makepsres* terminates with an error if it encounters a file it cannot identify.