Common Desktop Environment 1.0

Programmer's Guide

This edition of the Common Desktop Environment Advanced User's and System Administrator's Guide applies to AIX Version 4.2, and to all subsequent releases of these products until otherwise indicated in new releases or technical newsletters.

Part 1 —Basic Integration

Part 2 —Recommended Integration

Part 3 —Optional Integration

About This Book

Who Should Use This Book

Use this book if you are a programmer interested in integrating an existing application into the Common Desktop Environment (CDE), or in developing a new application that uses the features and functionality of CDE. This book describes the CDE development environment, and assumes that you are familiar with Motif®, X, UNIX®, or C programming.

Before You Read This Book

The *Common Desktop Environment: Programmer's Guide* is a collection of programming information. The manuals listed in the section "Related Books" should be read before you begin integration of any applications to CDE.

The *Common Desktop Environment: Programmer's Overview* provides a description of CDE and introduces the programming environment.

How This Book Is Organized

The *Common Desktop Environment: Programmer's Guide* has two parts. Each part provides a detailed description of each element of the Common Desktop Environment, a conceptual diagram, and a task-oriented description of how to use each element, complete with code examples.

Part 1 – "Basic Integration" introduces how to register your application and printing levels.

"Basic Application Integration" describes the steps involved with the basic integration of an existing application into CDE.

Part 2 – "Recommended Integration" introduces how to integrate existing applications into the Common Desktop Environment.

"Integrating Fonts" describes how to use generic standard font descriptions to ensure that you get the closest matching font for your application on any CDE-compliant system.

"Displaying Errors from Your Application" describes a common model for presenting information and error messages.

"Integrating with Session Manager" describes the ICCM session management protocol and provides examples of how to integrate your application with Session Manager.

"Integrating with Drag and Drop" describes the drag-and-drop user model, the new drag-and-drop application program interface (API), and how to use drag and drop.

Part3 – "Optional Integration" describes how to integrate new applications with the Session Manager and with drag and drop. It also explains how locales affect the Login Manager, Window Manager, and the terminal emulator.

"Integrating with the Workspace Manager" describes how to integrate your application with the Workspace Manager in specialized ways.

"Common Desktop Environment Motif Widgets" describes how to use the custom widgets that are provided as part of CDE.

"Invoking Actions from Applications" describes how to create actions within your application.

"Accessing the Data-Typing Database" describes the data-typing functions and how to use the data-typing database.

"Integrating with Calendar" introduces the Calendar API, including functions, data structures, calendar attributes, and entry attributes. It also describes how to use the Calendar API.

"Glossary" is a list of words and phrases found in this book and their definitions.

Related Books

Before beginning integration of your application into CDE, you should become familiar with the other books in the documentation set. See "Development Environment Documentation" for a list of the companion books.

The run-time environment documentation set consists of:

- *Common Desktop Environment: User's Guide*
- *Common Desktop Environment: Advanced User's and System Administrator's Guide*
- Online help volumes

Note: The *Advanced User's and System Administrator's Guide* contains information to help you integrate an application into the desktop

For more information about the Calendaring and Scheduling API, contact the X.400 API Association for the latest copy of the XAPIA Specification. The address is X.400 API Association, 800 El Camino Real, Mountain View, California, 94043.

Development Environment Documentation

This section provides an overview of each manual—except for the *Programmer's Guide*—in the developer documentation set. In addition to the *Programmer's Guide*, the development environment documentation set consists of:

- *Common Desktop Environment: Style Guide and Certification Checklist*
- *Common Desktop Environment: Application Builder User's Guide*
- *Common Desktop Environment: Programmer's Overview*
- *Common Desktop Environment: Help System Author's and Programmer's Guide*
- *Common Desktop Environment: ToolTalk Messaging Overview*
- *Common Desktop Environment: Internationalization Programmer's Guide*
- *Common Desktop Environment: Desktop Korn Shell User's Guide*
- *Common Desktop Environment*: Glossary
- Online man pages

Common Desktop Environment: Programmer's Overview

The *Common Desktop Environment: Programmer's Overview* has two parts. Part 1 contains an architectural overview of the Common Desktop Environment, including high-level information on both the run-time and development environments. Part 2 contains information useful to know before developing an application, and describes the development environment components.

The *Common Desktop Environment: Programmer's Overview* provides a high-level view of the Common Desktop Environment development environment and the developer documentation set. Read this book first before starting application design and development.

Common Desktop Environment: Style Guide and Certification Checklist

The *Common Desktop Environment: Style Guide and Certification Checklist* provides application design style guidelines and the list of requirements for Common Desktop Environment application-level certification. These requirements consist of the Motif Version 1.2 requirements with Common Desktop Environment-specific additions.

The checklist describes keys using a model keyboard mechanism. It assumes that your application is being designed for a left-to-right language environment in an English-language locale. Wherever keyboard input is specified, the keys are indicated by the engravings on the Motif model keyboard. Mouse buttons are represented using a virtual button mechanism to specify behavior independent of the number of buttons on the mouse.

This book provides information to assist the application designer in developing consistent applications and behaviors within the applications.

Common Desktop Environment: Application Builder User's Guide

The Common Desktop Environment Application Builder (also called *App Builder*) is an interactive tool for developing Common Desktop Environment applications. AppBuilder provides features that facilitate both the construction of an application graphical user interface (GUI) and the incorporation of the desktop's many useful desktop services (such as Help, ToolTalk, and Drag and Drop). The Common Desktop Environment: Application Builder User's Guide explains how to create an interface by dragging and dropping "objects" from a palette. The guide also explains how to make connections between objects in the interface, use the application framework editor to easily integrate desktop services, generate C code, and add application code to the App Builder output to produce a finished application.

Common Desktop Environment: Help System Author's and Programmer's Guide

The *Common Desktop Environment: Help System Author's and Programmer's Guide* describes how to develop online help for application software. It covers how to create help topics and integrate online help into a Motif application.

The audience for this book includes:

- Authors who design, create, and view online help information
- Developers who want to create software applications that provide a fully integrated help facility

This book has four parts. Part 1 describes the collaborative role that authors and developers undertake to design application help. Part 2 provides information for authors organizing and writing online help. Part 3 describes the Help System application programmer's toolkit. Part 4 contains information for both authors and programmers about preparing online help for different language environments.

Common Desktop Environment: ToolTalk Messaging Overview

The *Common Desktop Environment: ToolTalk Messaging Overview* describes the ToolTalk components, commands, and error messages offered as convenience routines to enable your application to conform to Media Exchange and Desktop Services message set conventions. This manual is for developers who create or maintain applications that use the ToolTalk® service to interoperate with other applications.

The T*oolTalk Messaging Overview* does *not* describe general ToolTalk functionality. For detailed information about the ToolTalk service, refer to *The ToolTalk Service: An Inter-Operability Solution*. For tips and techniques to help make using ToolTalk easier, read *ToolTalk and Open Protocols: Inter-Application Communication*.

Common Desktop Environment: Internationalization Programmer's Guide

The *Common Desktop Environment: Internationalization Programmer's Guide* provides information for internationalizing an application so that it can be easily localized to support various languages and cultural conventions in a consistent user interface.

Specifically, this guide:

- Provides guidelines and hints for developers on how to write applications for worldwide distribution.
- Provides an overall view of internationalization topics that span different layers within the desktop.
- Provides pointers to references and more detailed documentation. In some cases, standard documentation is referenced.

This guide is not intended to duplicate the existing reference or conceptual documentation, but rather to provide guidelines and conventions on specific internationalization topics. It focuses on internationalization topics and not on any specific component or layer in an open software environment.

Common Desktop Environment: Desktop Korn Shell User's Guide

The *Common Desktop Environment: Desktop Korn Shell User's Guide* describes how to create Motif applications with Desktop Korn Shell (**dtksh**) scripts. It contains several example scripts of increasing complexity, in addition to the basic information a developer needs to get started.

This guide is intended for developers who find a shell-style scripting environment suitable for a particular task. It assumes a knowledge of Korn Shell programming, Motif, the Xt Intrinsics, and, to a lesser extent, Xlib.

Common Desktop Environment: Glossary

The *Common Desktop Environment: Glossary* provides a comprehensive list of terms used in the Common Desktop Environment. The Glossary is the source and reference base for all users of the desktop. Because the audience for this glossary consists of many different types of users—from end users to developers to translators—the format for a glossary definition may include information about the audience, where the term originated, and the Common Desktop Environment component that uses the term in its graphical user interface.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your .login file. Use ls -a to list all files. system% You have mail.
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type rm filename.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Section 6 in User's Guide. These are called class options. You must be root to do this.

Typographic Conventions		
Typeface or Symbol	Meaning	Example
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	`system%`
$	UNIX Bourne and Korn shell prompt	`system$`
#	Superuser prompt, all shells	`system#`

Part 1 —Basic Integration

Basic Application Integration

Basic application integration is a set of highly recommended tasks you should perform.

- Basic Integration Features
- Basic Integration Tasks
- Complete Print Integration
- Partial Print Integration
- Nonintegrated Printing

Basic integration does not involve extensive use of the desktop application programmer's interface (API). Therefore, it does not provide other interaction with the desktop, such as drag and drop, session management, ToolTalk messaging, and programmatic access to the actions and data typing database.

A few of the integration tasks covered in this section require source code modification. They are optional, and are discussed here because they are closely related to basic integration tasks.

Basic Integration Features

Basic application integration provides these features for end users:

- A graphical way to locate and start your application on the desktop

Your application will provide a desktop registration package, and your installation script will automatically register your application.

Registration creates an application group at the top level of Application Manager. The application group contains an icon the user double–clicks to start the application.

- The ability to recognize and manipulate your application's data files

Your application will provide data types for its data files.

Data typing configures data files to use a unique icon to help users identify them. The data files also have meaningful desktop behavior. For example, the user can start your application by double–clicking a data file; dropping a data file on a desktop printer drop zone prints the file using the appropriate print command.

- Easy font and color selection using Style Manager

Your application will change interface fonts and background, foreground, and shadow colors dynamically.

The desktop defines general interface font and color resources that are used if no corresponding application–specific resources exist.

Basic integration provides these advantages to system administrators:

- Easy installation and registration

Upon installation, the application is automatically registered. The system administrator has little or no additional work to do.

- Easy ongoing administration

All the desktop's configuration files are gathered in one location. Furthermore, the application can easily be unregistered if, for example, the administrator wants to update it or to move it to a different application server.

Organization of Basic Integration Information

Most of the tasks involved in basic integration are also performed by system administrators who are integrating an existing application into the desktop. Therefore, most basic integration documentation is located in the section "Registering an Application" in the *Common Desktop Environment: Advanced User's and System Administrator's Guide*.

This section guides you to that information and contains additional information specific to application programmers.

Basic Integration Tasks

These are the general tasks involved in basic integration:

- Modify any application resources that set fonts and colors. This allows users to change the application's interface fonts and colors using Style Manager.

See the section on modifying font and color resources in the section "Registering an Application" in the *Common Desktop Environment: Advanced User's and System Administrator's Guide*.

- Create the registration package for your application.

See the text, "Creating a Registration Package for Your Application and "Registering an Application" in the *Common Desktop Environment: Advanced User's and System Administrator's Guide.*

- Modify your application's installation script to install the registration package files and perform the registration procedure.

See the text on registering the application using **dtappintegrate** in the section "Registering an Application" in the *Common Desktop Environment: Advanced User's and System Administrator's Guide.*

- Print application data files on networked and local printers. The desktop printer model provides a graphical way for users to print and is built on top of the native networking capabilities of the UNIX **lp** service.

Levels of Printing Integration

The printing functionality available to the user depends on the level of integration you use. There are three levels of integration:

- Complete integration. See "Complete Print Integration."

You should do complete integration if you have the ability to modify the application's source code.

When you do complete print integration, users can print data files on various printers by dropping them on printer drop zones (the Front Panel Printer control and printer icons in Print Manager). Certain other desktop behaviors are also implemented (see "Desktop Printing Environment Variables").

- Partial integration. See "Partial Print Integration."

You should do partial integration if you do not have the ability to modify the application's source code, but you do have the ability to invoke printing via an action.

When you do partial integration, your application provides a subset of full–integration functionality. For example, by using the **LPDEST** environment variable, your application's printing mechanism will obtain the print destination from the drop zone.

- No integration. See "Nonintegrated Printing."

If an application can not supply a print action for its data files, you should configure the data files to display an error dialog box when users drop the files on printer drop zones.

Complete Print Integration

To do complete print integration, your application must:

- Provide a Print action
- Use (dereference) the four desktop printing environment variables

Desktop Printing Environment Variables

To have fully integrated printing, your application must dereference the following four environment variables. The LPDEST variable is particularly important. It provides the ability for the user to choose the print destination by using a particular printer drop zone.

Printing Variable	Description
LPDEST	Uses the specified value as the printer destination for the file. If the variable is not set, the default printing device for your application should be used.
DTPRINTUSERFILENAME	Specifies the name of the file as it should appear in the Print dialog or print output. If the variable is not set, the actual file name should be used.
DTPRINTSILENT	Specifies whether to display a Print dialog box. When the variable is set to True, the Print dialog should not be displayed. If the variable is not set, the dialog box should be displayed.
DTPRINTFILEREMOVE	When the variable is set to True, the file should be removed after it is printed. This functionality is intended for temporary files that don't need to be retained after printing is complete. If the variable is not set, the file should not be removed.

A Fully Integrated Print Action

The Print action is part of the registration package, provided in a configuration file, *app_root*/**dt/appconfig/types/**/*language*/*name*.**dt**.

If your print action executes a program that dereferences the four environment variables indicated in "Desktop Printing Environment Variables," then your data type is fully integrated. The Print action must be written to be specific for the application's data type and should accept only a single file.

For example, the following print action is specific for a data type named **ThisAppData**:

```
ACTION Print

{
ARG_TYPEThisAppData
EXEC_STRING print_command –file %(file)Arg_1%
}
```

If your application handles the Print ToolTalk request, then your print action could send a variant of it with the following actions. (If any of the four environment variables are not set,

the corresponding message argument will be null. When the message argument is null, refer to “Desktop Printing Environment Variables” for the default interpretation.)

```
ACTION Print

{

        ARG_TYPEThisAppData

        ARG_CLASSFILE

        ARG_COUNT1

        TYPE             TT_MSG

        TT_CLASSTT_REQUEST

        TT_SCOPETT_SESSION

        TT_OPERATIONPrint

        TT_FILE%Arg_1%

        TT_ARG0_MODETT_IN

        TT_ARG0_VTYPE%Arg_1%

        TT_ARG1_MODETT_IN

        TT_ARG1_VTYPELPDEST

        TT_ARG1_VALUE$LPDEST

        TT_ARG2_MODETT_IN

        TT_ARG2_VTYPEDTPRINTUSERFILENAME

        TT_ARG2_VALUE$DTPRINTUSERFILENAME

        TT_ARG3_MODETT_IN

        TT_ARG3_VTYPEDTPRINTSILENT

        TT_ARG3_VALUE$DTPRINTSILENT

        TT_ARG4_MODETT_IN

        TT_ARG4_VTYPEDTPRINTFILEREMOVE

        TT_ARG4_VALUE$DTPRINTFILEREMOVE

}

ACTION Print

{

        ARG_TYPEThisAppData

        ARG_CLASSBUFFER
```

```
            ARG_COUNT1

            TYPETT_MSG

            TT_CLASSTT_REQUEST

            TT_SCOPETT_SESSION

            TT_OPERATIONPrint

            TT_ARG0_MODETT_IN

            TT_ARG0_VTYPE%Arg_1%

            TT_ARG0_VALUE%Arg_1%

            TT_ARG1_MODETT_IN

            TT_ARG1_VTYPELPDEST

            TT_ARG1_VALUE$LPDEST

            TT_ARG2_MODETT_IN

            TT_ARG2_VTYPEDTPRINTUSERFILENAME

            TT_ARG2_VALUE$DTPRINTUSERFILENAME

            TT_ARG3_MODETT_IN

            TT_ARG3_VTYPEDTPRINTSILENT

            TT_ARG3_VALUE$DTPRINTSILENT

            TT_ARG4_MODETT_IN

            TT_ARG4_VTYPEDTPRINTFILEREMOVE

            TT_ARG4_VALUEfalse

    }
```

Creating Print Actions for Filtered Data or Data Ready to Print

The desktop print utility **/usr/dt/dtlp** provides functionality on top of the **lp** subsystem. It gathers **lp** print options and prints the specified file.

Your application can use dtlp if either of the following conditions are true:

- The data files do not need to be processed before they are sent to a printer.
- Or, your application provides a filter for converting its data files to a ready–to–print form.

For more information about **dtlp**, see the **dtlp(1)** man page.

If the file is ready to print, the Print action runs **dtlp** in the **EXEC_STRING**. For example:

```
Print
{
        ARG_TYPE          ThisAppData
        EXEC_STRING       dtlp %Arg_1%
}
```

If the application provides a conversion filter, the filter must be run before running **dtlp**. For example:

```
Print
{
         ARG_TYPE         MyAppData
         EXEC_STRING      /bin/sh 'cat %Arg_1%| filter_name | dtlp'
}
```

where *filter_name* is the name of the print filter.

Partial Print Integration

To do partial print integration, your application must provide:

- A Print action
- The extent to which printing is integrated depends on which, if any, of the printing environment variables are handled by the action.

Providing the Print Command for Partial Integration

To provide partial print integration, your application must provide a print command line of the form:

```
print_command [options]–file filename
```

where *options* provides a mechanism for dereferencing none, some, or all of the printing environment variables (see "Desktop Printing Environment Variables").

The simplest form of this print command line omits options.

```
print_command  –file filename
```

This command line lets users print your application's data files using the desktop printer drop zones. However, printing destination is not set by the drop zone. In addition, other print behaviors set by the environment variables are not implemented. For example, the desktop may not be able to direct silent printing or remove temporary files.

If your print command line provides additional command–line *options* that correspond to the desktop printing environment variables, you can provide additional integration.

For example, the following command line provides the ability to dereference **LPDEST**:

```
print_command [–d destination] [–file filename]
```

where:

destination is the destination printer.

The next print command line provides options for dereferencing all four variables:

```
print_command [–d destination] [–u user_file_name] [–s] [–e] –file
filename
```

where:

user_file_name
: The file name as seen by the user.

–s Printing is silent (no Print dialog box is displayed).

–e The file is removed after it is printed.

The dereferencing occurs in the action definition. See the section, “Desktop Printing Environment Variables” for more information.

Turning Environment Variables into Command Line Switches

If your action is not capable of dereferencing the four environment variables, but it is capable of taking corresponding command line options, this subsection explains how to turn the environment variable values into command line options.

For example, this is a simple Print action that deferences **LPDEST**:

```
Print
{
        ARG_TYPE          data_type
        EXEC_STRING       print_command –d $LPDEST –file %(file)Arg_1%
}
```

However, this Print action may create unpredictable behavior if **LPDEST** is not set.

One way to create a Print action that provides proper behavior when variables are not set is to create a shell script that is used by the Print action.

For example, the following action and the script it uses properly handle all four environment variables:

```
Print
        ARG_TYPE        data_type
        EXEC_STRING     app_root/bin/envprint %(File)Arg_1%
}
```

The contents of the **envprint** script follows:

```
#!/bin/sh
# envprint – sample print script
DEST=""
USERFILENAME=""
REMOVE=""
SILENT=""

if [ $LPDEST ] ; then
        DEST="–d $LPDEST"
fi

if [ $DTPRINTUSERFILENAME ] ; then
        USERFILENAME="–u $DTPRINTUSERFILENAME"
fi

DTPRINTFILEREMOVE=echo $DTPRINTFILEREMOVE | tr "[:upper:]"
"[:lower:]"‘
if [ "$DTPRINTFILEREMOVE" = "true" ] ; then
        REMOVE="–e"
fi

DTPRINTSILENT=‘echo $DTPRINTSILENT | tr "[:upper:]" "[:lower:]"‘
if [ "$DTPRINTSILENT" = "true" ] ; then
        SILENT="–s"
fi

print_command $DEST $USERFILENAME $REMOVE $SILENT –file $1
```

Nonintegrated Printing

If your application does not integrate printing with the desktop, users must open your application to properly print data files.

Nevertheless, you should provide a Print action that runs when users drop your application's data files on a printer drop zone. Otherwise, the desktop may assume that the file contains text data, and the print output will be garbled.

The desktop provides a print action for this purpose named NoPrint. The NoPrint action displays a dialog box telling users that the data files cannot be printed using the printer drop zones.

The NoPrint action displays the Unable to Print dialog box.

To use the Unable to Print dialog box, create a print action specific to your data type that maps to the NoPrint action. For example, suppose the data type for your application is:

```
DATA_ATTRIBUTES MySpreadSheet_Data1
{
        _
}
```

The following Print action maps to the NoPrint for this data type:

```
ACTION Print
{
        ARG_TYPE        MySpreadSheet_Data1
        TYPE            MAP
        MAP_ACTION      NoPrint
}
```

Creating a Registration Package for Your Application

The desktop registration package you create for an application should become part of the application's installation package. The procedures for creating a registration package are also performed by system administrators integrating existing applications into the desktop. These procedures are documented in "Registering an Application" in the *Common Desktop Environment: Advanced User's and System Administrator's Guide*.

Part 2 —Recommended Integration

Integrating Fonts

Your application may be used by someone sitting at an X terminal, or by someone at a remote workstation across a network. In these situations, the fonts available to the user's X display from the X window server might be different from your application's defaults, and some fonts may not be available.

The standard font names defined by CDE are guaranteed to be available on all CDE-compliant systems. These names do not specify actual fonts. Instead, they are aliases that each system vendor maps to its best available fonts. If you use only these font names in your application, you can be sure of getting the closest matching font on any CDE-compliant system.

- Using Fonts in CDE Configuration Files
- Default Font Name
- Point Size
- Standard Application Font Names in **app-defaults** files

Using Fonts in CDE Configuration Files

CDE specifies a set of generic standard application font names, in several sizes, that can be used by applications running under CDE on all platforms. Each CDE vendor maps the standard set of font names to its available fonts. The mapping of font names to existing fonts may vary from vendor to vendor.

When you use the standard application font names in your app-defaults files, you can use a single app-defaults file across all CDE platforms. If you do not use the standard font names, you must supply a different app-defaults files for each application on each CDE platform.

All CDE systems provide a set of 13 standard application font names, in at least 6 sizes, that represent 12 generic design and style variations (serif and sans serif), as well as a symbol font. These standard names are provided in addition to the names of the fonts that the standard names are mapped to for a particular CDE platform. An additional four standard font names—to allow both serif and sans serif designs in a monospaced font—may also be provided by CDE platform vendors, if they choose to do so.

These 13 font names are provided in CDE platforms for the locales using the ISO 8859–1 character set. See the *Common Desktop Environment: Internationalization Programmer's Guide* for information on using standard font names in other locales.

Default Font Names

The set of font names is defined by the XLFD field name values described in the following table:

Field Name Values for Font Names		
Field	Value	Description
FOUNDRY	dt	CDE name
FAMILY_NAME	application	CDE standard application font name
WEIGHT_NAME	medium or bold	Weight of the font
SLANT	r i	Roman Italic
SET_WIDTH	normal	Normal set width
ADD_STYLE	sans serif	Sans serif font Serif font
PIXEL_SIZE	*	Platform dependent
POINT_SIZE	*pointsize*	Point size of the desired font
RESOLUTION_X	*	Platform dependent
RESOLUTION_Y	*	Platform dependent
AVERAGE_WIDTH	p m	Proportional Monospace
NUMERIC FIELD	*	Platform dependent
CHAR_SET_REGISTRY	iso8859-1	Defining standards authority
ENCODING	1	Character set number

The standard names are available using the regular X Windows XLFD font-naming scheme. When properly specified with appropriate wildcards for the platform-dependent fields, a CDE font name is guaranteed to open a valid, corresponding platform-dependent font. The XLFD name returned from a call to the Xlib **XListFont** function, however, is not guaranteed to be the same on all CDE platforms.

Using these values, the XLFD pattern

```
-dt-application-*
```

matches the full set of CDE standard application font names on a given platform. The pattern

```
-dt-application-bold-*-*-*-*-*-*-*-p-*-*-*-
```

matches the bold, proportionally spaced CDE fonts, both serif and sans serif. And the pattern

```
-dt-application-*-*-*-*-*-*-*-*-m-*-*-*-
```

matches the monospaced fonts (whether serif or sans serif, or both).

The full set of CDE Standard Application Font Names can be represented as follows:

```
-dt-application-bold-i-normal-serif-*-*-*-*-p-*-iso8859-1
-dt-application-bold-r-normal-serif-*-*-*-*-p-*-iso8859-1
-dt-application-medium-i-normal-serif-*-*-*-*-p-*-iso8859-1
-dt-application-medium-r-normal-serif-*-*-*-*-p-*-iso8859-1
-dt-application-bold-i-normal-sans-*-*-*-*-p-*-iso8859-1
-dt-application-bold-r-normal-sans-*-*-*-*-p-*-iso8859-1
-dt-application-medium-i-normal-sans-*-*-*-*-p-*-iso8859-1
-dt-application-medium-r-normal-sans-*-*-*-*-p-*-iso8859-1
-dt-application-bold-i-normal-*-*-*-*-*-m-*-iso8859-1
-dt-application-bold-r-normal-*-*-*-*-*-m-*-iso8859-1
-dt-application-medium-i-normal-*-*-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-*-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-*-*-*-*-p-*-dtsymbol-1
```

Point Sizes

The complete set of point sizes available for each of the standard application font names is determined by the set of fonts shipped with a vendor's CDE platform, whether bitmapped only or both bitmapped and scalable outline. The minimum set of sizes required and available on all CDE platforms corresponds to the standard sizes of bitmapped fonts that make up the default mapping for X11R5: 8, 10, 12, 14, 18, and 24.

For example, the entire set of six sizes of the plain monospaced font can be represented by the patterns:

```
-dt-application-medium-r-normal-*-80-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-100-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-120-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-140-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-180-*-*-*-m-*-iso8859-1
-dt-application-medium-r-normal-*-240-*-*-*-m-*-iso8859-1
```

These patterns match the corresponding standard font name on any CDE platform, even though the numeric fields other than **POINTSIZE** may be different on various platforms, and the matched fonts may be either serif or sans serif, depending on how the vendor implemented the set of standard names.

Standard Application Font Names in app-defaults files

You can code a single **app-defaults** file to specify font resources for your application and use it across all CDE platforms. Because the parts of the standard names that are defined are the same across different vendors' platforms, you can specify these values in the resource specification in the **app-defaults** file. However, you must use wildcards for the other fields (**PIXEL_SIZE**, **RESOLUTION_X**, **RESOLUTION_Y**, and **AVERAGE_WIDTH**) because they may vary across platforms. For example, to specify some of the default resource needs for an application named `appOne`, you might use:

```
appOne*headFont: \
-dt-application-bold-r-normal-sans-*-140-*-*-p-*-iso8859-1

appOne*linkFont: \
-dt-application-bold-i-normal-sans-*-100-*-*-p-*-iso8859-1
```

As another example, suppose that `appTwo` running on a vendor's platform defines two font resources for headings and hypertext links. `appTwo` uses a 14 point bold, serif font (Lucidabright bold) and a 12–point bold, italic sans serif font (Lucida bold–italic). You would then change the font definition from:

```
apptwo *headingFont: \
-b&h-lucidabright-bold-r-normal--20-140-100-100-p-127-iso8859-1

apptwo *linkFont: \
-b&h-lucida-bold-i-normal-sans-17-120-100-100-p-96-iso8859-1
```

to:

```
apptwo *headingFont: \
-dt-application-bold-r-normal-serif-*-140-*-*-p-*-iso8859-1

apptwo *linkFont: \
-dt-application-bold-i-normal-sans-*-120-*-*-p-*-iso8859-1
```

in your **app-defaults** file. Even though you may not know the names of the fonts on other CDE platforms, these platform-independent patterns specified with the CDE standard application font names match appropriate fonts on each platform.

You encode them exactly as shown, complete with the * wildcards, in your resource definitions. By applying the wildcards to the numeric fields other than point size, you ensure that the resources match CDE fonts on all platforms, even if the exact pixel size or average width of the fonts is slightly different.

See the *Common Desktop Environment Programmer's Reference* for more information.

Displaying Errors from Your Application

Users running your application expect messages to be displayed in message footers, error dialogs, or warning dialogs, with further explanations available in online help when appropriate. Applications in the Common Desktop Environment follow a common model for presenting error messages and warnings.

- How to Present Error Messages
- Information to Present in Error Dialogs
- Linking Message Dialogs to Online Help
- Recovery Routines

How to Present Error Messages

Because of the way message text is handled, users may not see messages from your application unless you display them in a dialog, footer, or elsewhere in the user interface.

In CDE, such messages are directed to log files that a casual user may not routinely examine. Use the following rules when deciding where to tell users about warnings, messages, and error conditions:

- If the message is informational**,** display the text in the message footer of the application; for example, “MyDoc file copied.”
- If the message is about an error or serious warning—a problem where an operation important to the user has failed—display an error dialog or warning dialog.

Information to Present in Error Dialogs

A good error dialog or warning dialog gives the user the following information:

- What happened (from the user’s point of view)
- Why it happened, in simple language that the user can relate to the current task and environment
- How to fix the problem

If the information cannot be presented in four or five lines of the error dialog, consider adding a help button to the dialog and link it to a topic in the help volume for your application.

For more information on writing messages, see the *Common Desktop Environment: Internationalization Programmer’s Guide*.

Linking Message Dialogs to Online Help

In cases where additional background information is required, or where it takes more than four or five lines of a dialog to explain an error fully, you should add a button that links the user to online help.

Adding online help for a dialog is a straightforward task. Once you have decided that a particular dialog is a candidate for online help, do the following:

1.. Choose a unique ID for the error help.

   This ID provides the link to the online help text. IDs should be 64 characters or less; for example, `DiskSpaceError.`

2.. Create the dialog and add a help callback.

Use the **XmCreateErrorDialog** convenience function for error messages and **XmCreateWarningDialog** for warnings, adding the help callback as follows:

```
XtAddCallback(dialog, XmNhelpCallback, helpfn, "ID");
```

In this example, helpfn is a help function you have created to manage the help dialog, and the string "ID" is the ID you chose for the error message (for example, `DiskSpaceError`). In your help function, set the **XmNlocationId** resource to the value of `ID.` The **/usr/dt/examples/dthelp** directory contains examples of how to set up such a help function.

For detailed information about creating and managing help dialog widgets, see the *Common Desktop Environment: Help System Author's and Programmer's Guide*.

3.. Write a corresponding help section for the error message.

Document the message in the "messages" section of your help volume. In the help source document, you should have a separate section for each message, and the ID= attribute at the beginning of the section should match the ID you chose in your code for the error.

For example, in the `s1` section heading, the `ID` is `DiskSpaceError.`

When the user's system has insufficient disk space, the error message the user sees from the following heading is "Could Not Save File."

```
<s1 ID=DiskSpaceError>Could Not Save File <\s1>
```

Note that by convention, the text of the section heading should correspond closely to the text in the error dialog.

4.. Rebuild the help file.

The new help section for the error message becomes active as soon as you rebuild the help file (using the `dthelptag` program) and recompile your application.

For information about writing and building online help, see the *Common Desktop Environment: Help System Author's and Programmer's Guide*.

Recovery Routines

If a recovery routine exists for an error condition, consider adding a Retry button to the dialog. For example, if a file could not be copied because the system had insufficient disk space, you might offer a Recopy option in the dialog that users could choose once they have corrected a disk space or permissions problem.

Integrating with Session Manager

Session Manager saves information about the Desktop environment and the applications running when the user logs out (of the current session) or when the user saves the environment (in a home session). For an application to be saved as part of the current session or the home session and then restarted as part of the next session, it must participate in the X Inter–Client Communication Conventions Manual (ICCCM) 1.1 Session Management Protocol. This section outlines how Session Manager saves and restores sessions and details the steps necessary for an application to participate in session management.

- How Session Manager Saves Sessions and Applications
- How to Program the Application for Session Management
- How Session Manager Restores a Session

How Session Manager Saves Sessions and Applications

When you exit a session or when you save a Home session, Session Manager:

1.. Saves the selected resource settings and X server settings

2.. Allows each application to save its state and waits for the save to be completed

3.. Obtains the command line required to restart the application

How to Program the Application for Session Management

Setting the Program Environment

This section describes the programming steps necessary for an application to be saved as part of the integration process.

Follow these steps to set the program environment:

1.. Include the following header files:

- **Xm/Xm.h**
- **Xm/Protocols.h**
- **Dt/Session.h**

2.. Link with **libXm** and **libDtSvc**.

3.. Initialize the toolkit and create a top-level widget.

Setting the WM_SAVE_YOURSELF Atom

Use the Motif **XmAddWMProtocol()** function to set the **WM_SAVE_YOURSELF** atom on the **WM_PROTOCOLS** property for the top-level window of your application, as shown in the following example.

```
Atom XaWmSaveYourself;
Display *dsp;

dsp = XtDisplay(toplevel);
XaWmSaveYourself =
XmInternAtom(dsp, ”WM_SAVE_YOURSELF”, False);
XmAddWMProtocols(toplevel, &XaWmSaveYourself, 1);
```

Note: Do not set the **WM_SAVE_YOURSELF** atom for more than one window.

Prepare to Receive the WM_SAVE_YOURSELF Message

Use the Motif **XmAddWMProtocolCallback()** function to establish a callback procedure to be called when the application receives a **WM_SAVE_YOURSELF** client message:

```
XmAddWMProtocolCallback(toplevel, XaWmSaveYourself,
SaveYourselfProc,
   toplevel);
```

Processing the WM_SAVE_YOURSELF Message

When Session Manager sends a **WM_SAVE_YOURSELF** client message to this sample application's top-level window, the **SaveYourselfProc()** callback procedure is called. Use thecallback to save the application's state. The application can save its state by any means you want, but cannot interact with the user during the save.

Session Manager provides the **DtSessionSavePath()** function as a way to return a full path name and a base file name to use for saving the application's state.

Setting the WM_COMMAND Property

After the application has finished processing the **WM_SAVE_YOURSELF** message, either by saving its state or ignoring the message, the application must set the **WM_COMMAND** property on its top-level window to tell Session Manager that the save operation is complete.

Use the **Xlib XsetCommand()** function to set the **WM_COMMAND** property on the application's top-level window. Setting this property lets Session Manager know that the application has finished processing the **WM_SAVE_YOURSELF** message and gives Session Manager the command line it needs to restart the application.

XsetCommand() accepts an array of command-line arguments. If the application uses the **DtSessionSavePath()** function as part of the save process, **XsetCommand()** needs an additional command-line argument: –session basename, where basename is the base file name returned by **DtSessionSavePath()**.

How Session Manager Restores a Session

Session Manager restores a session by:

1.. Restoring the resource database and server settings

2.. Restarting applications using the saved command lines

If the application used **DtSessionSavePath()** to find a path for its saved state, the application can pass the base file name from the –session argument to the **DtSessionRestorePath()** function to find the full path name of its saved-state file.

Integrating with Drag and Drop

This section describes the drag-and-drop user model and the Common Desktop Environment drag-and-drop convenience application program interface (API), and describes how to use drag and drop.

- Summary
- Drag-and-Drop User Model
- Drag-and-Drop Convenience API
- Drag-and-Drop Transaction
- Integration Action Plan
- API Overview
- How Drag Sources Are Used
- How Drop Zones Are Used

Summary

The Common Desktop Environment contains an application program interface (API) for drag and drop that is layered on top of Motif to provide convenient, consistent, and interoperable drag and drop across the desktop. The Common Desktop Environment drag-and-drop API makes it easier for developers to implement drag and drop. With drag and drop, users can manipulate objects on the screen directly by grabbing them, dragging them around the display, and dropping them on other objects to perform a transfer of data.

Text, files, and buffers are the three categories of data that are used with the Common Desktop Environment drag-and-drop API. Text is defined, in this context, as any user-visible text such as text in type-in fields. A file is a container of data that resides in the file system. Each file also has a format that describes its contents. Buffers are data contained in memory. Typically, each buffer also has a format that describes its contents.

Library and Header Files

To use drag and drop, you need to link to the **DtSvc** library. The header file is **Dt/Dnd.h .**

Demo Program

A demo program containing an example of drag and drop is in **/usr/dt/examples/dtdnd**.

Using Drag and Drop

To Integrate with Drag and Drop

To integrate your application with drag and drop, follow these steps:

1.. Include **Dt/Dnd.h**.

2.. Link to **libDtsvc .**

3.. As recipient:

   a. Register as a drop zone using **DtDndDropRegister**.

   b. Optionally, write a drop animate callback.

   c. Write a transfer callback.

4.. As source:

a. Recognize user action (possibly requiring a modification of translation tables) and call **DtDndDragStart**.

b. Write **a convert callback**.

c. Write **a drag finish callback**.

Drag-and-Drop User Model

This section describes the user model behind drag and drop to help you design an application that is consistent with the rest of the desktop and users' expectations.

See the *Common Desktop Environment: Style Guide* for more information about the drag-and-drop user model and for guidelines for the visual appearance of drag-and-drop elements.

When drag and drop is available for all applications on the desktop, the system is more predictable to the user and is, therefore, easier to use and to learn. Users leverage their learning across more applications by using skills that they already know. In addition, many users prefer drag and drop to using menus.

In this section, the term *drop zone* is used to describe places where users can drop something. Drop zones are usually represented by a control or icon graphic; for example, a trash icon or a type-in field graphic. The term drop target is used to describe the rectangular area that represents the drop zone.

Drag and Drop Capability

With the Drag and Drop capability, users can select and manipulate objects represented as icons.

Note: Drag and drop is an accelerator to functionality that is accessible through other user interface controls supported within your application. However, not all users are able to take advantage of drag and drop. Do not support any basic operations solely through drag and drop. Any basic function that your application supports through drag and drop should also be supported by menus, buttons, or dialog boxes.

Drag Icons

When users select and manipulate icons using drag and drop, they expect the graphic icon that represents the item being dragged to remain consistent from the selection through the drag and drop. If the user selects a message icon in the File Manager and starts to drag it, the source portion of the drag icon is represented by that message icon. Providing this kind of consistency makes drag and drop more predictable to the user. Where the destination application uses icons, the icon shown should, in most cases, be the same one that was selected and then dragged and dropped. This behavior is not, however, always appropriate for all applications. Dragging text is an exception. A text drag icon is used instead of dragging the selected text.

Both the source and destination applications specify the visual appearance of drag icons. You are responsible for ensuring that an application has a consistent and appropriate icon to drag. Although the drag-and-drop library provides default icons, it is a good idea for you to specify your own for each application. Most often, you should use the data-typing database to obtain the icon associated with the type data represented by the icon. See "Accessing the Data-Typing Database".

When users start a drag without selecting an icon, it is appropriate for you to provide a relevant drag icon. For example, in an appointment editor, the user can select an appointment out of a scrolling list—which may or may not show icons. You should use an appointment icon as the source indicator. The destination application (for example, a File Manager) should display the same appointment icon.

Parts of the Drag Icon

The drag icon changes appearance to provide *drag-over feedback* when the user moves it over potential drop zones.

The drag icon has three parts that combine to provide the drag-over feedback:

- A state indicator
- An operation indicator
- A source indicator

The state indicator is a pointer used for positioning combined with a valid or invalid drop zone indicator. The valid state indicator is an arrow pointer. The pointer has a hot spot so users can position it in a predictable manner. The invalid state indicator—a circle with a diagonal line—is displayed when users have positioned the cursor over an invalid drop zone.

The operation indicator gives users feedback on what operation is occurring during the drag; either move, copy, or link. Because most drags are moves, users are given additional feedback when they perform the less-frequent copy or link operations.

Note: The operation feedback is drawn on top of the state and source feedback. This behavior is consistent with Motif drag-and-drop behavior.

The user can choose the drag operation move, copy, or link by pressing and holding certain keys during a drag, as shown in the following table:

Keys Used to Modify a Drag Operation	
Modifier Key	Operation
Shift	Move
Control	Copy
Control and Shift	Link

The source application can force a copy, as in the case of the read-only File Manager window. When the user chooses an operation, the drop zone must match that operation for the drop to succeed; otherwise, the drop zone is invalid. In other words, if the user chooses a copy by holding down the Control key, and then drags the drag icon over the trash icon, the drag icon should show the trash icon as an invalid drop zone and any drop should fail, because copying to the trash is not allowed.

The source indicator represents the selection (or the item being dragged). The source indicator varies depending on whether the selection represents single or multiple items and what kind of item the selection represents. The following table shows the default source indicators in the Common Desktop Environment. These source indicators are generated automatically when you use the Common Desktop Environment drag-and-drop convenience API. The icons are approximations, not exact screen representations.

Types of Drag Icons

Drag Icons	Text Selection	Single Selection	Multiple Selection
Valid Move			
Valid Copy			
Valid Link			
Invalid Move None			
Invalid Copy			
Invalid Link			

Drags from Inside Windows

Sometimes an application needs to enable a drag from within a dialog box or window. The Appointment Editor in Calendar has a scrolling list of appointments and an entry area for editing an appointment. Users can drag from the scrolling list to get an appointment, but users also need to be able to drag from the appointment entry area. Enabling users to drag from the entry area covers those times when the appointment is not yet inserted in the calendar (for example, when a proposed meeting time is entered but not inserted into the calendar).

The item that can be dragged needs to have an icon graphic associated with it. Place the icon graphic in the dialog box in an appropriate area adjacent to the information to be dragged. The upper-right corner of the dialog box or window is the recommended default position. The icon lets the user know that something can be dragged and the graphic used is the same graphic used in the drag icon to provide consistency. The icon should be 32x32 pixels and have a label so that it resembles a File Manager icon. See the *Common Desktop Environment: Style Guide* section on drag and drop for more information.

Note: Drags are only enabled from human interface elements that have components or items that can be selected. Drags cannot be enabled from static labels such as those on buttons or menus.

Visual Feedback

The following sections describe the drop zone feedback and transition effects of drag and drop.

Drop Zone Feedback

The default drop zone feedback, called *drag under*, can be a solid line drawn around the site, a raised or lowered surface with a beveled edge around the drop zone, or a pixmap drawing over the drop zone.

Transition Effects

Transition effects show the user that the drop has either succeeded or failed. The two transition effects are *melt* and *snap back.*

Melting occurs when the user drops a drag icon on a valid drop zone. When the user drops a drag icon on a valid drop zone, the drag icon melts into the drop zone. The drag icon is replaced by the icon appropriate to the destination application. A printer on the Front Panel may show nothing other than the melting effect. An open File Manager window may display an appropriate icon.

When an icon is dropped, sometimes the melting effect does not take place immediately. The icon is displayed where it is placed until the transfer is done. It is a good idea for the destination to set its cursor to a busy state while the transfer is occurring. The user cannot move or select the icon until the transfer is complete; the busy cursor lets the user know the transfer is in process.

Snap back occurs when a drop fails. Drops can fail in two ways. If the user drops a drag icon over an invalid drop zone, then the drag icon snaps back to the source application. Once a drop occurs, the source and destination applications have to transfer the data. If the data transfer fails, the drag icon snaps back and the destination application is responsible for indicating failure to the user and providing information on why the drop failed.

Drag-and-Drop Sources

To help you understand the behavior of drag-and-drop sources, the following table describes the key desktop components that can be a source of drags of text selections, files, and buffers.

	Desktop Components That Can Be Drag Sources		
Drag Source	Text Selections	Files	Buffers
Text fields (Motif)*	Selected text	N/A	N/A
Text Editor: Main Window	Selected Text	N/A	N/A
Terminal: Main Window	Selected Text	N/A	N/A
File Manager: Folder Window	N/A	Files	N/A
File Manager: Trash Window	N/A	Files	N/A
Mail: Message List	N/A	N/A	Message in mail-message format
Mail: Attachment List	N/A	N/A	Attachment in format of the attachment
Calendar: Appointment Editor	N/A	N/A	Appointment in appointment format

*Any application that has Motif text field sources selected drags text.

Drag-and-Drop Destinations

The following components on the desktop provide drop destinations:

- Editors
- File Manager
- Front Panel

Each component accepts drops of text selections, files, and buffers. Most of the text drop destinations are provided automatically by the Motif library. File or buffer data drop destinations require additional code.

When a user drops data from a file, and that file is modified in some way, the modifications can be written back to the original holder of the file. This behavior is described as saveback. However, when data is dropped from a buffer, the data does not have information about an originating file. As a result, changes to data from buffers cannot be written back, because there is no original holder of the data. This behavior is described as no saveback.

For example, the Mailer can export mail attachments to editors using drag and drop. If the attachment is exported as a buffer (that has no saveback), the editor has no way to change the original attachment in the mailer. So, the editor can only save its modified version of the attachment to a new file.

Because mail attachments are not already separate files (they are embedded into a mail folder file), they are only exported as buffers and cannot be saved back by other editors.

If the attachment is exported as a file (that has saveback) the editor saves its modified version to that same file.

The following table describes the drops of text selections, files, and buffers on editor-type components such as Text Editor, Icon Editor, Calendar, Mailer, and Application Builder.

Editor Drop Destinations			
Drop Destination	Text Selections	Files	Buffers
Text Editor: Main Window	Insert	Insert	Insert
Terminal: Main Window	Insert	N/A	N/A
Icon Editor: Main Window	N/A	Load (if file in icon format) saveback	Load into read-only (if data in icon format) no saveback
Mailer: Message List	N/A	Append (if file in mail format)	Append (if data in mail format)
Mailer: Compose	Insert	Insert	Insert
Mailer: Attachment List	Attach	Attach	Attach
Calendar: Main Window	N/A	Schedule Appointment (if file in appointment format)	Schedule appointment (if data in appointment format)
Calendar: Appointment Editor	Insert into text field	Fill in appointment fields (if file in appointment format)	Fill in appointment fields (if data in appointment format)
AppBuilder	N/A	Load (if file in BIX or BIL format) saveback	Load into read-only (if data in BIP format) no saveback

The following table describes the drops of text selections, files, and buffers on file and folder icons in the File Manager.

File Manager Drop Destinations			
Drop Destination	Text Selections	Files	Buffers
File Icon	Invoke drop action on target file and dropped text (if file accepts text drops and dropped text in appropriate format) no saveback/copy	Invoke drop action on target file and dropped file (if file accepts file drop and dropped file in appropriate format) saveback	Invoke drop action on target file and dropped data (if file accepts data drop and dropped data in appropriate format) no saveback/copy
Folder Icon	Insert text into new file using “Untitled” name in folder	Copy/move file to folder	Insert data into new file using supplied name (if available) in folder else using “Untitled”
Action Icon	Invoke action on text (if appropriate format and accepts text drop) no saveback	Invoke action on files (if appropriate format and accepts file drop) saveback	Invoke action on data (if appropriate format and accepts data drop) no saveback
Mail Container Icon	Append to mailbox (if text in mail format)	Append to mailbox (if file in mail format)	Append to mailbox (if data in mail format)

The following table describes the drops of text selections, files, and buffers on action icons in the Front Panel.

Front Panel Drop Destinations			
Drop Destination	Text Selections	Files	Buffers
Text Editor	Load into read-only no saveback	Load saveback	Load into read-only no saveback
Calendar	Schedule appointment (if text in appointment format)	Schedule appointment (if file in appointment format)	Schedule appointment (if data in appointment format)
Mail	Compose message attach text	Compose message attach file	Compose message attach data
Printer	Print text (if print method available for text)	Print file contents (if print method available for file format)	Print data (if print method available for data format)
Trash Can	N/A	Move file to Trash Can	N/A

	Front Panel Drop Destinations		
Drop Destination	Text Selections	Files	Buffers
Subpanel: Install Icon	N/A	Install icon	N/A
Subpanel: Action	Same as File Manager	Same as File Manager	Same as File Manager
Subpanel: Executable	Same as File Manager	Same as File Manager	Same as File Manager

See the *Common Desktop Environment: Style Guide* for more information and guidelines on how the drag and drop should appear to the user.

Drag-and-Drop Convenience API

The Common Desktop Environment provides a drag-and-drop convenience API to promote consistency and interoperability across the desktop, and to make it easier for developers to implement drag and drop.

The existing Motif API for drag and drop provides reasonable functionality to achieve a rendezvous between the source and destination applications in the transaction. It provides a framework for data transfer but leaves the actual data transfer details up to the application. For true consistency and interoperability between applications across the desktop, all applications must use the same data transfer protocols. The Common Desktop Environment drag-and-drop convenience API provides common data transfer routines.

Simplify Use for Developers

The existing Motif API for drag and drop is very flexible and, therefore, is somewhat difficult for nonexpert developers to use. The Common Desktop Environment drag-and-drop convenience API provides some convenience functions, described in the following paragraphs, that result in an API that is simpler and easier to use by providing the following services:

- Manages configuration and appearance of drag icons. Graphics are provided for the default source, state, and operation icons that make up the drag icon in Motif. The compositing of these icons checks the type of data being dragged.
- Enables animation for a drop. You can define an animation procedure that is called when the drop has completed.
- Provides data transfer using standard X selection targets for text, file, and buffer transfers. This data transfer allows interoperability with other applications that use the standard targets directly.
- Provides dual registration. You can register a text widget as a drop zone for data other than text and preserve the ability to accept text drops.

Establish Policy

The drag-and-drop API establishes policy in three areas:

- Common targets. Where available, existing selection targets defined by the Inter-Client Communication Conventions Manual (ICCCM) are used.
- Data transfer protocols. The API hides some of the details of data transfer and presents the data to the application in the form of some simple data structures.

- Default drag icons. Default drag icons are provided for applications that can accept them.

Provide Common Functionality

The drag-and-drop API provides common functionality in these areas:

- Supports the transfer of data as text, file names, and buffers
- Supports, through the data transfer framework, the addition of new, built-in protocols

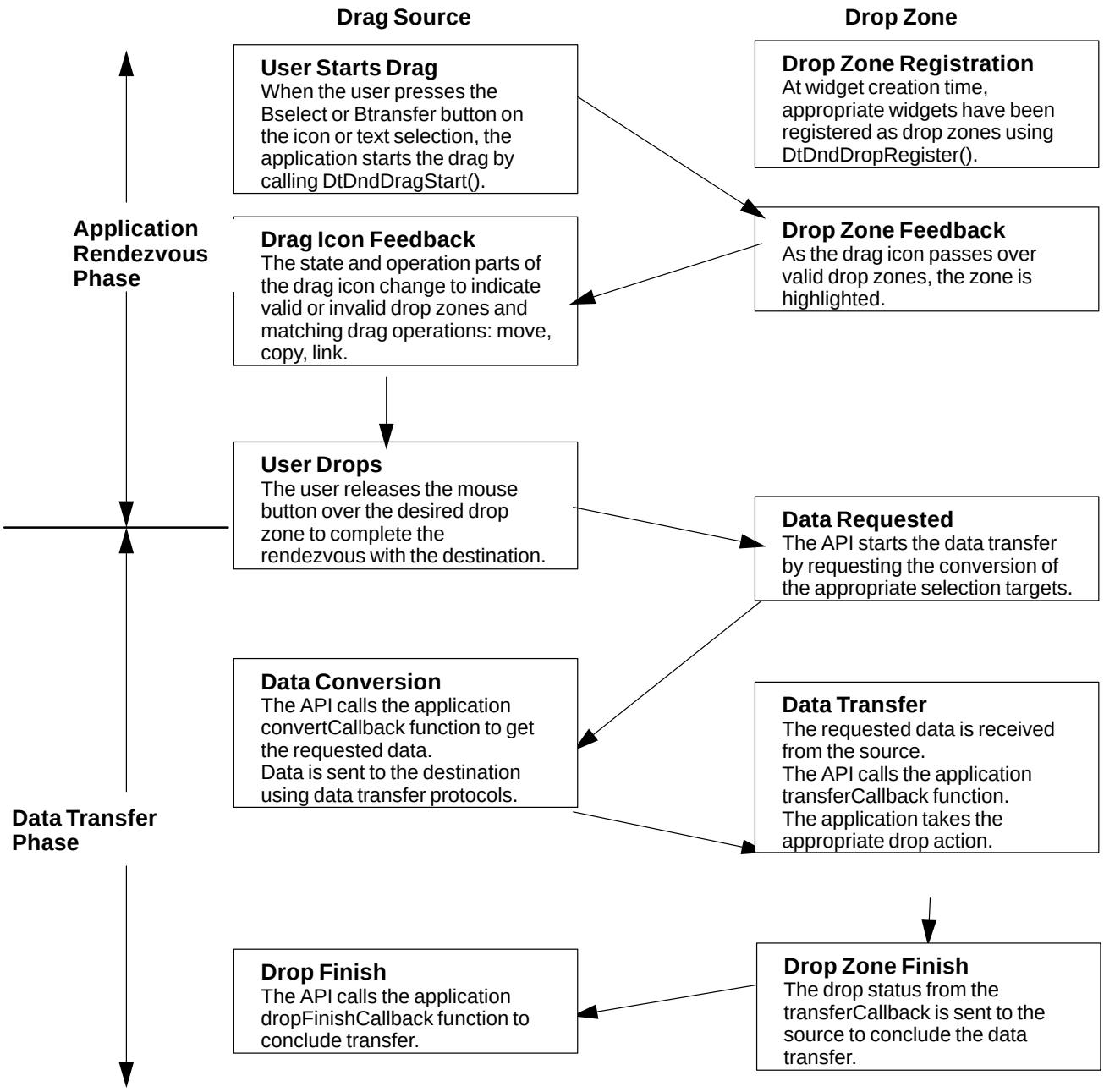
Leverage Existing Motif API

The API for drag and drop does not invent a new drag-and-drop subsystem; rather, it uses the existing Motif API. In addition, since common data transfer protocols were chosen, where available, applications can interoperate at the selection protocol level without requiring global use of the new API.

The transfer of text and files use existing protocols. Buffer transfer uses new protocols.

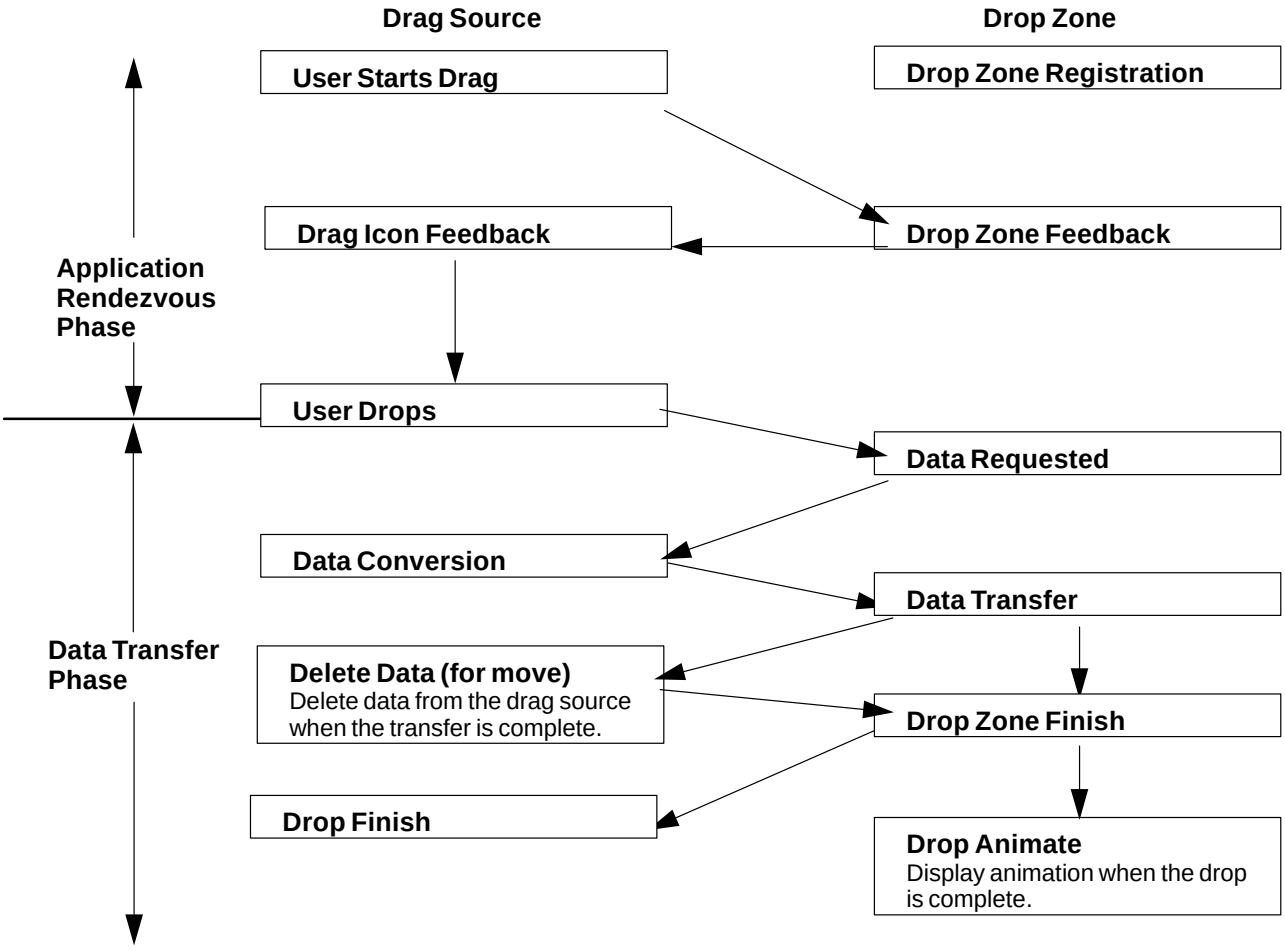
Drag-and-Drop Transaction

The Basic Drag-and-Drop Transaction figure illustrates how the basic drag-and-drop transaction is performed. The dotted-line boxes show the basic transaction. The solid boxes show the optional transitions and operations.



The Basic Drag-and-Drop Transaction

The Optional Transitions and Operations for Drag and Drop figure illustrates the optional transitions and operations for the drag-and-drop transaction.



Optional Transitions and Operations for Drag and Drop

Integration Action Plan

This section suggests a plan of action for integrating your application with drag and drop in Common Desktop Environment 1.0.

Review Drag-and-Drop API and Sample Code

Use the information provided in this section and in the relevant sections in the *Common Desktop Environment: Programmer's Reference* to familiarize yourself with the drag-and-drop API. Once you have a basic understanding of the API, review the source code for the drag-and-drop demo program, **/usr/dt/examples/dtdnd**. This code provides examples of how to use the API in various ways. The examples should give you an understanding of the character and amount of code you need to write to support drag and drop in your application. Understanding the actions and the data-typing API is useful as well.

Review Your Application for Possible Drop Zones

Identify the types of data your application might accept through a drag-and-drop transaction. If, for example, you are writing a bitmap editor, you want to support the drop of files. Once you have identified the data types you will allow to be dropped on your application, determine the widget or widgets that should be drop zones. For the bitmap editor example, you may decide the only place a file should be dropped on the application is the bitmap

editing area. In this case, register the widget representing this area using **DtDndDropRegister()** and provide the appropriate callbacks.

Because it is easiest to handle the drop of file names, start by implementing them. Once you have mastered this technique, you will find it easier to move on to implementing the drop of text and buffers.

Review Your Application for Possible Drag Sources

Identify the types of data your application might permit as sources for a drag-and-drop transaction. In the example of the bitmap editor, you may want bitmap data containing the current bitmap selection to be a drag source as an accelerator for cut and paste. Once you have identified the data types you will allow to be dragged from your application, determine the widget or widgets that should be drag sources. In the bitmap editor example, you may decide the bitmap editing area containing the highlighted bitmap selection should serve as the drag source. In this case, enable the widget representing this area for sourcing drags.

Start by implementing the drag of buffers that are most appropriate or specific to your application. You will also want to add the ability to drop buffers on your application to enable easy data transfer between multiple invocations of your application.

API Overview

This section provides an overview of the drag-and-drop application program interface (API).

DtSvc Library and Header File

The drag-and-drop functionality is implemented in the Desktop Services library, **DtSvc**. To access the drag-and-drop API, include the header file **<Dt/Dnd.h>** and link with -**lDtSvc**.

Functions

The API includes four function calls, which are declared in the header file Dnd.h and outlined in the following paragraphs. These functions are described in greater detail in later sections.

- **DtDndDragStart()** starts a drag in response to a user action.
- **DtDndCreateSourceIcon()** creates a source icon to use with **DtDndDragStart()**.
- **DtDndDropRegister()** registers a widget as a drop zone. Drop zone registration usually occurs immediately after the widget is created, but may be performed at any time.
- **DtDndDropUnregister()** unregisters a previously registered widget. A drop zone is usually unregistered immediately before being destroyed, but may be unregistered any time after being registered.

The DtDndContext Structure

You handle transfers of data using the **DtDndContext** data structure. This structure contains fields for the transfer protocol, the number of items being transferred, and an array of the data items being transferred. See the **DtDndDragStart(3X)** and **DtDndDropRegister(3X)** man pages for details about the syntax of this structure.

Protocols

Protocols are used to tell the API the type of data being transferred. The predefined protocols are shown in the following table.

Predefined Protocols	
Protocol	Description
DtDND_TEXT_TRANSFER	Text transfer. Compound text. (Motif uses a compound text target for text transfers.)
DtDND_FILENAME_TRANSFER	File name transfer.
DtDND_BUFFER_TRANSFER	Memory buffer.

Operations

The drag source and the drop zone can transfer the data in one of three ways, as described in the following table.

Data Transfer Operations	
Operation	Description
XmDROP_MOVE	Moves the data (Copy followed by Delete).
XmDROP_COPY	Copies the data.
XmDROP_LINK	Contains a link to the data.

How Drag Sources Are Used

This section describes how drag sources are used.

Starting a Drag

A drag is started in one of two ways. First, the user may start a drag by pressing down **Btransfer**, the middle mouse button. As soon as the button is pressed down, the drag begins. Second, the user may start a drag by pressing and holding down **Bselect**, the left mouse button, and moving the cursor across the screen. The drag begins when the user moves the mouse a certain distance. This distance is called the drag threshold and it is measured in pixels. The default drag threshold is 10 pixels for **Bselect**. For **Btransfer**, the drag threshold is 0; because there is no drag threshold, the drag begins as soon as the pointer is moved. Motif scrolled text lists and text widgets are automatically registered as drag sources for text drags using **Btransfer** and **Bselect**.

Dragging from Lists or Icons

There are two common interface objects that can be used to source a drag: lists and icons. The Motif List widget automatically sources text drags. If other types of drags are desired, this is accomplished by overriding the default widget translations with new **Bntl** and **Btn2** translations. There is no icon widget in Motif but often a drawing area is used as container of icons. In this case, an event handler for **Btn1Motion** would be used to start the drag. Refer to the sample code in **/usr/dt/examples/dtdnd** for more detailed code examples.

Drag Threshold

When starting a drag using **Bselect** the widget event handler or translation procedures must apply the drag threshold of 10 pixels before starting the drag. For **Btransfer**, there is no threshold and the drag starts immediately.

Btransfer or Badjust

Style Manager has a setting in the Mouse category that controls whether **Btn2**, the middle mouse button, acts as **Btransfer** or **Badjust**. This setting is stored as a resource name: **enableBtn1Transfer**. A setting of 1 indicates that **Btn2** is **Badjust** and should adjust the selection while a setting of any other value means that **Btn2** is Btransfer and should start a drag. **Btn1**, the left mouse button, always starts a drag.

The following example shows how to determine whether **Btn2** should be **Btransfer** or **Badjust**.

```
Display* display;
int adjust;

XtVaGetValues ((Widget)XmGetXmDisplay(display,
        “enableBtn1Transfer”, &adjust,
        NULL);

if (adjust == 1)
        /* Btn2 is adjust */
        else
      /* Btn2 is transfer */
```

Initiating a Drag

Common Desktop Environment 1.0 applications start a drag by calling **DtDndDragStart()**. This function performs some desktop-specific setup and calls **XmDragStart()** to initiate a drag. The **DtDndDragStart()** function synopsis and parameter usage are described as follows:

```
Widget
DtDndDragStart(
     Widget             dragSource,
     XEvent         *event,
     DtDndProtocol   protocol,
     Cardinal        numItems,
     unsigned char   operations,
     XtCallbackList  convertCallback,
     XtCallbackList  dragFinishCallback
     ArgList      argList,
     Cardinal     argCount)
    Widget       dragSource
```

The widget that received the event that triggered the drag.

`XEvent *event`

The button press or button motion event that triggered the drag.

`DtDndProtocol protocol`

The protocol used for the data transfer. The protocol may be one of the following:

```
DtDND_TEXT_TRANSFER
DtDND_FILENAME_TRANSFER
DtDND_BUFFER_TRANSFER
```

`Cardinal numItems`

Specifies the number of items being dragged.

`unsigned char operations`

Specifies options supported by dragSource. The options are **XmDROP_MOVE**, **XmDROP_COPY**, and **XmDROP_LINK**. A drag source may support any combination of these operations. You specify a combination of operations by using or. For example, to support the move and copy operations, specify `XmDROP_MOVE | XmDROP_COPY`.

`XtCallbackList convertCallback`

This callback is invoked when a drop has started and the drop zone has requested data from the drag source. The **convertCallback** is explained in more detail in the next section.

`XtCallbackList dragFinishCallback`

This callback is invoked when the drag-and-drop transaction is complete. The **dragFinishCallback** should reset the **dragMotionHandler()** and free any memory allocated by the drag source during the drag-and-drop transaction.

Using Convert Callbacks

The convert callback provides data to the drop zone when a drop occurs. The first action in the convert callback is a verification of the reason field in the **callData**. If the reason is not **DtCR_CONVERT_DATA** or **DtCR_CONVERT_DELETE**, you should return immediately; otherwise, proceed to convert the data. For example, if you are handling the conversion of a file name, retrieve the appropriate file name from your internal data structures and copy it into the file data object. If your drag source supports the move operation, you need to support conversion of the **DELETE** target. That is, when **convertCallback** is called with a reason of **DtCR_CONVERT_DELETE**, perform the appropriate deletion action for the data that was moved. In the case of the file transfer, delete the file. Here is a simple **convertCallback** that handles the conversion and deletion of file names.

```
void
convertFileCallback(
Widget dragContext,
    XtPointer clientData,
    XtPointer callData)
{
    DtDndConvertCallbackStruct *convertInfo =
    (DtDndConvertCallbackStruct*) callData;
    char     *fileName = (char *) clientData;

    if (convertInfo->reason == DtCR_DND_CONVERT_DATA) {
              convertInfo->dragData->data.files[0]=
                                XtNewString(fileName);
} else if (convertInfo->reason == DtCR_DND_CONVERT_DELETE) {
              deleteFile(fileName);
} else {
              convertInfo->status = DtDND_FAILURE;
  }
}
```

How Drop Zones Are Used

This section describes how drop zones are used.

Registering a Drop Zone

You generally register drop zones just after the widget that is going to be the drop zone is created. If you want a *modal* drop zone, you may register the widget as a drop zone when you want users to be able to drop on it and unregister it when you do not want users to drop on it.

Motif text widgets are automatically registered as drop zones for text when they are created. Dual registration is allowed. If you want a text widget to accept drops of other data, such as file names, in addition to text, you may register the text widget as a drop zone for file names as well. The text drop functionality provided by Motif is preserved. The functionality for file-name (or other data-type) drops is layered on top.

Use the function **DtDndDropRegister()** to register a widget as a drop zone. This function handles dual registration, if necessary, performs desktop-specific setup, and calls **XmDropSiteRegister()**. The **DtDndDropRegister()** function synopsis and parameter use are as follows.

```
void DtDndDropRegister(
       Widget         dropSite,
       DtDndProtocol  protocols;
       unsigned char  operations;
       XtCallbackList transferCallback;
       ArgList        argList;
       Cardinal       argCount)
Widget dropSite
```

The widget that is being registered as a drop zone.

`DtDndProtocol protocols`

Specifies the list of data transfer protocols that the drop zone can use. To specify the use of more than one protocol, use OR with the protocol values.

`unsigned char operations`

The operations supported by the drop zone. The drop zone may support any combination of **XmDROP_MOVE**, **XmDROP_COPY**, and **XmDROP_LINK** by using OR for the desired combination of operations.

`XtCallbackList transferCallback`

This function accepts the data that is dropped on the drop zone. The transfer callback is explained in greater detail in the next section.

`ArgList argList`

Specifies an optional argument list.

`Cardinal argCount`

Specifies the number of arguments in **argList**.

Using the Transfer Callback

The transfer callback accepts data from the drag source when a drop occurs. The first action in the transfer callback is a verification of the reason field in the **callData**. If the reason is not **DtCR_DND_TRANSFER_DATA**, you should return immediately; otherwise, proceed with data transfer based on its type and the operation specified in the reason. For example, if you are handling the copy of a file, retrieve the file name from the data structure, open the file, and copy its contents. If your drop zone supports more than one data type, you need to support the transfer of each data type appropriately.

Here is a simple transfer callback for a drawing area drop zone that supports the copying of text and file-name data types.

```
void
        TransferCallback(
        Widget widget,
        XtPointer clientData,
        XtPointer callData)
{
    DtDndTransferCallbackStruct *transferInfo =
                (DtDndTransferCallbackStruct*) callData;
    int ii;
    DtDndcontext * dropData = transferInfo->dropData;
                return;
    switch dropData->protocol {
        case DtDND_FILENAME_TRANSFER:
                for (ii=0; ii < dropData->numItems; ii++) {
                drawTheString(dropData->data, strings[ii]);
                }
                break;
        case DtDND_TEXT_TRANSFER:
                for (ii=0; ii<dropData->numItems; ii++){
                drawTheFile(dropData->data.files[ii]);
             }
        break;
        default:
        transferInfo->status = DtDND_FAILURE;
    }
}
```

Using Data Typing

In an application that accepts drops of buffers, you may want to handle the dropped data in a different way depending on its type. To accomplish data typing, use the data-typing API. Data-typing function calls of interest are **DtDtsBufferToDataType()** and **DtDtsBufferToAttributeValue()**. The former returns the data attribute name for the data, the latter returns the value of a specified attribute of the data. Attributes you may find useful for drag and drop are shown in the following table.

Data–Typing Attributes

Attributes	Description
`ICON`	Path of icon to use for this data.
`MEDIA`	The Message Alliance media name for this data.

See “Accessing the Data-Typing Database” for more information.

Part 3 —Optional Integration

Integrating with the Workspace Manager

The Workspace Manager provides the means for an application to manage its windows within the desktop's multiple workspace environment. An application can perform four major tasks by communicating with the Workspace Manager:

- Place the application's windows in one or more workspaces
- Identify the workspaces in which the application's windows are located
- Prevent the application's windows from moving to another workspace
- Monitor changes to the workspaces, such as when a user switches from one workspace to another

Normally, Session Manager will get your application main window into the right workspace without your intervention. However, if your application has multiple top-level windows, you should use the Workspace Manager API to figure out where your windows are and save this data as part of your session state.

See "Integrating with Session Manager" for details on saving application-related information between sessions.

- Communicating with the Workspace Manager
- Placing an Application Window in Workspaces
- Identifying Workspaces Containing the Application Windows
- Preventing Application Movement Among Workspaces
- Monitoring Workspace Changes

Communicating with the Workspace Manager

An application communicates with the Workspace Manager by using functions provided by the desktop. These functions allow you to quickly and easily perform a variety of tasks associated with workspace management. The following is a list of these functions:

- **DtWsmAddCurrentWorkspaceCallback**
- **DtWsmAddWorkspaceFunctions**
- **DtWsmAddWorkspaceModifiedCallback**
- **DtWsmFreeWorkspaceInfo**
- **DtWsmGetCurrentBackdropWindows**
- **DtWsmGetCurrentWorkspace**
- **DtWsmGetWorkspaceInfo**
- **DtWsmGetWorkspaceList**
- **DtWsmGetWorkspacesOccupied**
- **DtWsmOccupyAllWorkspaces**
- **DtWsmRemoveWorkspaceCallback**

- **DtWsmRemoveWorkspaceFunctions**
- **DtWsmSetCurrentWorkspace**
- **DtWsmSetWorkspacesOccupied**

Segments of code from two demo programs (`occupy.c` and `wsinfo.c`) illustrate the use of these functions. Listings for `occupy.c`, `wsinfo.c`, and makefiles for several brands of workstations are in the directory **/usr/dt/examples/dtwsm**. See the applicable man page for more information on each function.

Placing an Application Window in Workspaces

An application can place its windows in any or all of the existing workspaces. DtWsmOccupyAllWorkspaces places the windows in all currently defined workspaces, while DtWsmSetWorkspacesOccupied places the windows in all workspaces named in a list that is passed to the function.

To Place an Application Window in All Workspaces

- Use **DtWsmOccupyAllWorkspaces**.

  In `occupy.c`, the callback **allWsCB** for the **Occupy All Workspaces** push button calls this function.

```
DtWsmOccupyAllWorkspaces (XtDisplay(toplevel),
                          XtWindow(toplevel));
```

  where:

- `XtDisplay(toplevel)` is the X display.
- `XtWindow(toplevel)` is the window to be placed in all workspaces.

See the **DtWsmOccupyAllWorkspaces** man page for more information on this function.

To Place an Application Window in Specified Workspaces

- Use **DtWsmSetWorkspacesOccupied**.

  In `occupy.c`, the callback **setCB** for the **Set Occupancy** push button calls this function.

```
DtWsmSetWorkSpacesOccupied (XtDisplay(toplevel),
                            XtWindow(toplevel), paWsSet, numSet);
```

  where:

- `XtDisplay(toplevel)` is the X display.
- `XtWindow(toplevel)` is the window to be placed in the workspaces.
- `paWsSet` is a pointer to a list of workspace names that have been converted to X atoms.
- `numSet` is the number of workspaces in the list.

See the **DtWsmSetWorkspacesOccupied** man page for more information on this function.

Identifying Workspaces Containing the Application Windows

The function **DtWsmGetWorkspacesOccupied** returns a list of the workspaces in which a specified application window resides. In `occupy.c`, the procedure **ShowWorkspaceOccupancy** calls this function. Based on the results of this call, **ShowWorkspaceOccupancy** changes the appearance of the toggle buttons that represent the workspaces. A check mark appears on every toggle button in whose workspace the application window resides.

To Identify Workspaces That Contain the Application Window

- Use **DtWsmGetWorkspacesOccupied**.

```
rval = DtWsmGetWorkspacesOccupied(XtDisplay(toplevel)
            XtWindow(toplevel), &paWsIn, &numWsIn);
```

where:

- `XtDisplay(toplevel)` is the X display.
- `XtWindow(toplevel)` is the window to be searched for in the workspaces.
- `paWsIn` is the address of a pointer to a list of workspace names that have been converted to X atoms.
- `numWsIn` is the address of an integer into which the number of workspaces in the list is placed.

After this call, loops are set up to compare the list of workspaces (found in the procedure **SetUpWorkspaceButtons** by **DtWsmGetWorkspaceList**) with the list of workspaces in which the application window was found to reside. The toggle button resource **XmNset** is set to True for each toggle button that represents a workspace in which the application window resides.

Preventing Application Movement Among Workspaces

The function **DtWsmRemoveWorkspaceFunctions** prevents an application from:

- Switching from one workspace to another
- Occupying all workspaces
- Being removed from the current workspace

DtWsmRemoveWorkspaceFunctions does this by making that portion of the desktop Workspace Manager (**dtwm**) window menu inactive. The application should call **DtWsmRemoveWorkspaceFunctions** before its top-level window is mapped because **dtwm** only checks workspace information at the time it manages the application's top-level window. If you need to call **DtWsmRemoveWorkspaceFunctions** after the application's top-level window is managed, then you must first call the **Xlib** function **XWithdrawWindow**, call **DtWsmRemoveWorkspaceFunctions**, and then call **XMapWindow** to remap the top-level window.

To Prevent Movement to Another Workspace

- Use **DtWsmRemoveWorkspaceFunctions**.

```
DtWsmRemoveWorkspaceFunctions(XtDisplay(toplevel),
                              XtWindow(toplevel));
```

where:

- `XtDisplay(toplevel)` is the X display.
- `XtWindow(toplevel)` is the window for which workspace movement is to be prevented.

Monitoring Workspace Changes

You can monitor workspace changes by using either or both of the following functions:

- `DtWsmAddCurrentWorkspaceCallback`
- `DtWsmWorkspaceModifiedCallback`

DtWsmAddCurrentWorkspaceCallback registers an application callback to be called whenever the Workspace Manager is switched to a new workspace. See the **DtWsmAddCurrentWorkspaceCallback(3)** man page for more information.

DtWsmWorkspaceModifiedCallback registers an application callback to be called whenever a workspace is added, deleted, or changed. See the **DtWsmWorkspaceModifiedCallback(3)** man page for more information.

To Monitor Workspace Switching

- Use **DtWsmAddCurrentWorkspaceCallback**.

  In the demo program `wsinfo.c`, this function is called after the top-level widget is realized.

  ```
  DtWsmAddCurrentWorkspaceCallback (toplevel, wschangecb, NULL);
  ```

  where

- `toplevel` is the application's top level widget.
- `wschangecb` is the name of the function to be called.
- `NULL` is the parameter for client data to be passed to the callback. In this case, no data is passed.

To Monitor Other Workspace Changes

- Use **DtWsmWorkspaceModifiedCallback**.

  ```
  DtWsmWorkspaceModifiedCallback (toplevel, wschangecb, NULL);
  ```

  where:

- `toplevel` is the application's top-level widget.
- `wschangecb` is the name of the function to be called.
- `NULL` is the parameter for client data to be passed to the callback. In this case, no data is passed.

Common Desktop Environment Motif Widgets

The Common Desktop Environment provides Motif (based on Motif 1.2) OSF patch-level 1.2.3 libraries (with bug fixes) and enhancements. In addition, the Common Desktop Environment provides four custom widgets you can use to provide certain OPEN LOOK™ and Microsoft® Windows functionality. This section describes these Motif custom widgets.

- Using Common Desktop Environment Motif
- Text Field and Arrow Button Widget (DtSpinBox)
- Text Field and List Box Widget (DtComboBox)
- Menu Button Widget (DtMenuButton)
- Text Editor Widget (DtEditor)

The widget library, libDtWidget, contains four widgets that combine or enhance functionality of existing Motif 1.2 widgets:

- **DtSpinBox** combines a text field and arrow buttons in a control that can increment or decrement numeric or text values.
- **DtComboBox** combines a text field and a list box in a control that displays one of the many valid choices for the text field.
- **DtMenuButton** provides menu-cascading functionality outside of the menu bar.
- **DtEditor** incorporates such single text editor functions as cut and paste.

These widgets provide common functionality across all Common Desktop Environment applications. None of these widgets support subclassing.

The Custom Widgets library depends directly on the following libraries:

- **Xm** library for the Motif superclass support
- **Xt** library for creation and manipulation of widgets
- **X11** Library for the base X Window System
- **DtSvc** for desktop support utilized by DtEditor

Using Common Desktop Environment Motif

The Common Desktop Environment Motif toolkit consists of the Motif 1.2 widget library with enhancements to existing functionality, bug fixes, and two new features.

Common Desktop Environment Motif adds functionality to the Motif 1.2.3 release while maintaining source compatibility. Common Desktop Environment Motif is source and binary compatible with Motif 1.2 applications. Existing Motif 1.2 applications will compile using Common Desktop Environment Motif. Existing Motif 1.2 binaries will run without modification using Common Desktop Environment Motif.

Common Desktop Environment Motif also provides four custom widgets not found in Motif 1.2. The custom widgets are described in detail in this section.

Motif Libraries

Use the Common Desktop Environment Motif and X11R5 libraries to develop a Common Desktop Environment Motif-compliant application for the X Window System. The Common Desktop Environment Motif libraries are the Motif 1.2.3 libraries (with bug fixes) and enhancements.

Motif Library (libXm)

Common Desktop Environment provides all the Motif 1.2 header files.

Motif UIL Library (libUil)

The Motif User Interface Language (UIL) is a specification language for describing the initial state of a Motif application's user interface.

Include the **uil/UilDef.h** header file to access UIL.

Motif Resource Manager Library (libMrm)

The Motif Resource Manager (MRM) creates widgets based on definitions contained in User Interface Definition (UID) files created by the UIL compiler. MRM interprets the output of the UIL compiler and generates the appropriate argument lists for widget creation functions. Use libMrm to access the Motif Resource Manager.

Include the **Mrm/MrmPublic.h** header files to access **libMrm** in your application.

Features Added to Motif

The following features have been added to Motif 1.2.3 to support desktop applications:

- Complete internationalization of toolkit error messages.
- **XmGetPixmap()** and **XmGetPixmapByDepth()** use the environment variable **XMICONSEARCHPATH** or **XMICONBMSEARCHPATH** as the icon search path. If neither of these variables is set, **XBMLANGPATH** is used, which governs the Motif 1.2 behavior**.**See Common Desktop Environment Motif man page for more information.

Enhancements to Existing Motif Functionality

The Common Desktop Environment **Xm** library contains minor enhancements to Motif to enable better usability by OPEN LOOK and Microsoft Windows users. The usability enhancements include:

- Enabling button two on a three-button mouse to be used to extend the current selection. This functionality is equivalent to the OPEN LOOK ADJUST mouse button.
- Enabling Tab to be used to move through a group of **PushButton** widgets and gadgets, **ArrowButton** widgets and gadgets, and **DrawnButton** widgets.
- Enabling button 3 to activate a **CascadeButton** menu.
- Providing three new resources (**pathMode, fileFilterStyle,** and **dirTextLabelStrin**g) for the **XmFileSelectionBox** widget, which gives it a new look and feel. See the **Common Desktop Environment: Programmer's Overview** for details about the new **XmFileSelectionBox** resources.
- Enabling interoperability with Microsoft Windows and OPEN LOOK by providing multiple key bindings for Motif virtual keys.

Each of the preceding enhancements can be controlled by a resource: either a widget resource (for **XmFileSelectionBox**) or an application-wide resource (all other cases). The default values for this resource provide behavior and an API that is identical to that of Motif 1.2. For information on these enhancements and resources, see the **XmDisplay(3x)** and **XmFileSelectionBox(3x)** man pages.

Visual Enhancements

Common Desktop Environment changes the Motif 1.2 look in the following ways:

- **RadioBox** fill color is changed to show state more clearly.
- **RadioBox** shape is changed from diamond to circular.

- A check glyph is added to the **CheckBox** to show state more clearly.
- **CascadeButtons** and menu items are changed to have an etched-in border when active.
- Thumb is removed from the read-only Scale to distinguish it from the Scale.
- Default shadow thickness is changed to 1 pixel.
- Default highlight thickness is changed to 1 pixel.
- Default **PushButton** visual draws the highlight inside the button's default shadow.

An application-wide resource can control each of these enhancements. The default values for these resources provide behavior and an API that is identical to that of Motif 1.2.

For information on these enhancements, see the **XmDisplay(3)**, **XmPushButton(3)**,**XmPushButtonGadget(3)**,**XmToggleButton(3)**, **XmToggleButtonGadget(3)**, and **XmScale(3)** man pages.

Text Field and Arrow Button Widget (DtSpinBox)

Use the **DtSpinBox** for cycling through a list of text items or incrementing and decrementing a numeric entry. **DtSpinBox** is a subclass of the **XmManager** class and is used to display a text field and two arrows.

The **DtSpinBox** widget is a user interface control used to increment and decrement an arbitrary text or numeric field. You can use it, for example, to cycle through the months of the year or days of the month. The following figure shows examples of the **DtSpinBox** widget.

Examples of the DtSpinBoc Widget

Library and Header Files

The **DtSpinBox** widget is in the **libDtWidget** library. The header file is **Dt/SpinBox.h**.

Demo Program

A demo containing an example of the **DtSpinBox** widget is in **/usr/dt/examples/dtwidget/controls.c**.

Compatibility with Motif 2.0

To use the Motif 2.0 widgets, change the Dt names for the class, types, and creation routines to **Xm**. For example, change all occurrences of **DtSpinBox** to **XmSpinBox** in your

code. This information is supplied in case you choose to port your application to Motif 2.0, but it is not a recommendation that you do so.

Note: Common Desktop Environment does not guarantee strict API or binary compatibility between the Common Desktop Environment widgets and the Motif 2.0 widgets.

Convenience Functions

The following table lists the convenience functions for the **DtSpinBox** widget. See the **DtSpinBox(3X)** man page for more information.

DtSpinBox Convenience Functions

Function	Description
`DtCreateSpinBox()`	Creates a SpinBox widget.
`DtSpinBoxAddItem()`	Adds an item into a DtSpinBox widget at a specified location.
`DtSpinBoxDeletePos()`	Deletes a specified item from a DtSpinBox widget.
`DtSpinBoxSetItem()`	Sets the current item in a DtSpinBox widget.

Classes

DtSpinBoxWidget inherits behavior and resources from the **Core**, **Composite**, **Constraint**, and **XmManager** classes.

The class pointer is **dtSpinBoxWidgetClass**.

The class name is **DtSpinBoxWidget**.

DtSpinBoxWidget does not support subclassing.

Resources

The **DtSpinBox** widget defines the following set of widget resources. The following table lists the class, type, default, and access for each resource.

- **DtNarrowLayout** specifies the style and position of the **DtSpinBox** arrows.
- **DtNarrowSensitivity** specifies the sensitivity of the arrows in the **DtSpinBox**.
- **DtNspinBoxChildType** specifies the style of the **DtSpinBox**.
- **DtNdecimalPoints** specifies the position of the decimal point within the integer value when the child type is numeric.
- **DtNincrementValue** specifies the amount to increment or decrement the position when the child type is numeric.
- **DtNinitialDelay** specifies the amount of time in milliseconds before the arrow buttons begin to spin continuously.
- **DtNnumValues** specifies the number of items in the **DtNvalues** list when the child type is a string.
- **DtNvalues** supplies the list of strings to cycle through when the child type resource is a string.
- **DtNmaximumValue** specifies the upper bound on the **DtSpinBox** range when the child type is numeric.

- **DtNminimumValue** specifies the lower bound on the **DtSpinBox** range when the child type is numeric.
- **DtNmodifyVerifyCallback** is called just before the **DtSpinBox** position changes. The application can use this callback to implement new application-related logic, including setting a new position, spinning to, or canceling the impending action.
- **DtNposition** has a different value based on the child type resource. When the child type is a string, **DtNposition** is the index into the **DtNvalues** list for the current item. When the child type is numeric, **DtNposition** is the integer value of the **DtSpinBox** that falls within the maximum and minimum value range.
- **DtNrepeatDelay** is the number of milliseconds between repeated calls to the **DtNvalueChangedCallback** while the user is spinning the **DtSpinBox**.
- **DtNvalueChangedCallback** is called whenever the value of the **DtNposition** resource is changed through the use of the spinner arrows.

See the **DtSpinBox(3X)** man page for more information.

DtSpinBoxWidge Resources

Name	Class	Type	Default	Access
DtNarrowLayout	DtCArrowLayout	unsigned char	DtARROWS_END	CSG
DtNarrowSensitivity	DtCArrowSensitivity	unsigned char	DtARROWS_SENSITIVE	CSG
DtNspinBoxChildType	DtCSpinBoxChildType	unsigned char	DtSTRING	CG
DtNdecimalPoints	DtCDecimalPoints	short	0	CSG
DtNincrementValue	DtCIncrementValue	int	1	CSG
DtNinitialDelay	DtCInitialDelay	unsigned int	250 ms	CSG
DtNnumValues	DtCNumValues	int	0	CSG
DtNvalues	DtCValues	XmStringTable	NULL	CSG
DtNmaximumValue	DtCMaximumValue	int	10	CSG
DtNminimumValue	DtCMinimumValue	int	0	CSG
DtNmodifyVerifyCallback	DtCCallback	XtCallbackList	NULL	C
DtNposition	DtCPosition	int	0	CSG
DtNrepeatDelay	DtCRepeatDelay	unsigned int	200 ms	CSG
DtNvalueChangedCallback	DtCCallback	XtCallbackList	NULL	C

Callback Structures

The callback structure follows and is described in the following table.

```
typedef struct {
        int      reason;
        XEvent   *event;
        Widget   widget;
        Boolean  doit;
        int      position;
        XmString value;
        Boolean  crossed_boundary;
} DtSpinBoxCallbackStruct;
```

DtSpinBox Callbacks

Callback	Description
reason	Use this callback for three possible reasons. For the first call to the callback at the beginning of a spin, or for a single activation of the spin arrows, DtCR_OK is the reason. If the DtSpinBox is being continuously spun, the reason is DtCR_SPIN_NEXT or DtCR_SPIN_PRIOR, depending on the arrow that is spinning.
event	A pointer to the event that caused this callback to be invoked. It can be NULL when the DtSpinBox is continuously spinning.
widget	The widget identifier for the text widget that is affected by the spin.
doit	Sets this value only when the call_data comes from the DtNmodifyVerifyCallback. For a modifyVerify callback, the setting of this field by an application determines whether the action that initiated the callback should be performed. When this field is set to False, the action is not performed.
position	The new value of the DtNposition resource that results from the spin.
value	The new XmString value displayed in the Text widget that results from the spin. This string must be copied if it is used beyond the scope of the call_data structure.
crossed_boundary	This Boolean is True when the spinbox cycles, and/or DtNspinBoxChildType of XmSTRING wraps from the first item to the last or the last item to the first. When the DtNspinBoxChildType is XmNUMERIC, the boundary is crossed when the DtSpinBox cycles from the maximum value to the minimum value or vice versa.

Example of DtSpinBox Widget

The following example shows how to create and use a **DtSpinBox** widget. You can find this code as part of the controls.c demo in the **/usr/dt/examples/dtwidget** directory.

```
/*
 * Example code for SpinBox
 */
```

```
#include <Dt/SpinBox.h>
static char *spinValueStrings[] = {
    “alpha”, “beta”, “gamma”, “delta”,
    “epsilon”, “zeta”, “eta”, “theta”,
    “iota”, “kappa”, “lambda”, “mu”,
    “nu”, “xi”, “omicron”, “pi”,
    “rho”, “sigma”, “tau”, “upsilon”,
    “phi”, “chi”, “psi”, “omega”
};

static void ModifyVerifyCb(Widget, XtPointer, XtPointer);

static void CreateSpinBoxes(Widget parent)
{
    Widget titleLabel, spinBox;
    XmString *valueXmstrings;
    int numValueStrings;
    XmString labelString;
    Arg args[20];
    int i, n;


    /* Create value compound strings */

    numValueStrings = XtNumber(spinValueStrings);
    valueXmstrings = (XmString *)XtMalloc(numValueStrings
    sizeof(XmString*));
    for (i = 0; i < numValueStrings; i++) {
    valueXmstrings[i] =
    XmStringCreateLocalized(spinValueStrings[i]);
    }
    /* Create title label */
    labelString = XmStringCreateLocalized(“SpinBox Widget”);
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    titleLabel = XmCreateLabel(parent, “title”, args, n);
    XtManageChild(titleLabel);
    XmStringFree(labelString);

    /*
    * Create a SpinBox containing string values.
    */

    n = 0;
    XtSetArg(args[n], DtNvalues, valueXmstrings); n++;
    XtSetArg(args[n], DtNnumValues, numValueStrings); n++;
```

```
    XtSetArg(args[n], DtNcolumns, 10); n++;
    spinBox = DtCreateSpinBox(parent, “spinBox1”, args, n);
    XtManageChild(spinBox);

    /*
    * Create a SpinBox containing numeric values to 3 decimal
places.
    * Position the arrows on either side of the displayed
value.
    */

    n = 0;
    XtSetArg(args[n], DtNspinBoxChildType, DtNUMERIC); n++;
    XtSetArg(args[n], DtNminimumValue, 1000); n++;
    XtSetArg(args[n], DtNmaximumValue, 100000); n++;
    XtSetArg(args[n], DtNincrementValue,1000); n++;
    XtSetArg(args[n], DtNdecimalPoints,3); n++;
    XtSetArg(args[n], DtNposition,1000); n++;
    XtSetArg(args[n], DtNarrowLayout,DtARROWS_SPLIT); n++;
    XtSetArg(args[n], DtNcolumns, 10); n++;
    spinBox = DtCreateSpinBox(parent, “spinBox2”, args, n);
    XtManageChild(spinBox);

    /*
    * Create a SpinBox containing numeric values to 2 decimal
places.
    * Position the arrows on the left of the displayed value.
    * Disallow alternate user changes by adding a
modify/verify callback.
    */

    n = 0;
    XtSetArg(args[n], DtNspinBoxChildType, DtNUMERIC); n++;
    XtSetArg(args[n], DtNminimumValue, 1500); n++;
    XtSetArg(args[n], DtNmaximumValue, 60500); n++;
    XtSetArg(args[n], DtNincrementValue,1500); n++;
    XtSetArg(args[n], DtNdecimalPoints,2); n++;
    XtSetArg(args[n], DtNposition,7500); n++
    XtSetArg(args[n], DtNarrowLayout,
DtARROWS_FLAT_BEGINNING); n++;
    XtSetArg(args[n], DtNcolumns, 10); n++;
    spinBox = DtCreateSpinBox(parent, “spinBox3”, args, n);
    XtManageChild(spinBox);

    XtAddCallback(spinBox, DtNmodifyVerifyCallback,
```

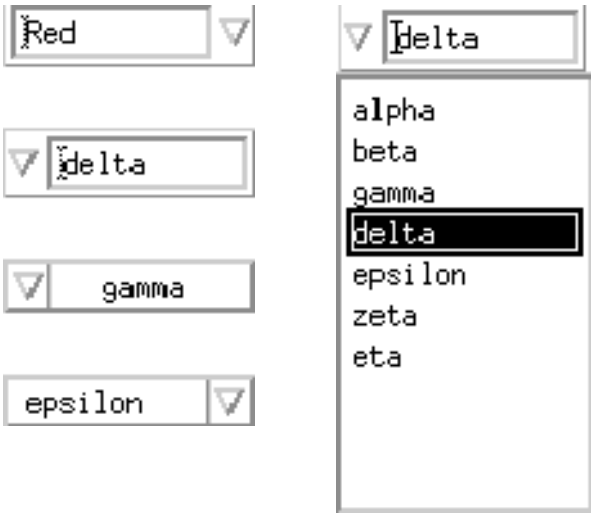
```
ModifyVerifyCb, NULL);

    /*
    * Create a SpinBox containing string values.
    * Position the arrows on the left of the display value
    */
    n = 0;
    XtSetArg(args[n], DtNvalues, valueXmstrings); n++;
    XtSetArg(args[n], DtNnumValues, numValueStrings); n++;
    XtSetArg(args[n], DtNarrowLayout, DtARROWS_BEGINNING);
n++;
    XtSetArg(args[n], DtNcolumns, 10); n++;
    spinBox = DtCreateSpinBox(parent, “spinBox4”, args, n);
    XtManageChild(spinBox);

   /*
   * Create a SpinBox containing numeric values to 3 decimal
places.
   * Position the arrows on the right of the displayed value.
   */

    n = 0;
    XtSetArg(args[n], DtNspinBoxChildType, DtNUMERIC); n++;
    XtSetArg(args[n], DtNminimumValue, 1000); n++;
    XtSetArg(args[n], DtNmaximumValue, 100000); n++;
    XtSetArg(args[n], DtNincrementValue,1000); n++;
    XtSetArg(args[n], DtNdecimalPoints,3); n++;
    XtSetArg(args[n], DtNposition,1000); n++;
    XtSetArg(args[n], DtNarrowLayout, DtARROWS_FLAT_END); n++;
    XtSetArg(args[n], DtNcolumns, 10); n++;
    spinBox = DtCreateSpinBox(parent, “spinBox5”, args, n);
    XtManageChild(spinBox);

    /*
    * Free value strings, SpinBox has taken a copy.
    */

    for (i = 0; i < numValueStrings; i++) {
    XmStringFree(valueXmstrings[i]);
    }
    XtFree((char*)valueXmstrings);

}
```

```
    /*
    * modify/verify callback.
    *
    * Allow/disallow alternate user changes

    */
static void ModifyVerifyCb(Widget w, XtPointer cd, XtPointer cb)
{
    DtSpinBoxCallbackStruct *scb= (DtSpinBoxCallbackStruct*)cb;
    static Boolean allowChange = True;

    scb–>doit = allowChange;

    if (allowChange == False) {
    printf("DtSpinBox: DtNmodifyVerifyCallback.
    Change disallowed.\n");
    XBell(XtDisplay(w), 0);
    }

    allowChange = (allowChange == True) ? False : True;
}
```

Text Field and List Box Widget (DtComboBox)

Use the **DtComboBox** widget to display a list and the current selection from the list. You can use this widget for display only, or as a selectable control.

The **DtComboBox** widget is a combination of a text field and a list widget that provides a list of valid choices for the text field. Selecting an item from this list automatically fills in the text field with that list item. The following figure shows examples of a **DtComboBox** widget.



Examples of text field and list box widget (DtComboBox)

Library and Header Files

The **DtComboBox** widget is in the **libDtWidget** library. The header file is **Dt/ComboBox.h**.

Demo Program

A demo containing an example of the **DtComboBox** widget is in **/usr/dt/examples/dtwidget/controls.c**.

Compatibility with Motif 2.0

The **DtComboBox** widget is similar to the Motif 2.0 release of **XmComboBox**. The APIs are designed so that an application can easily switch to the Motif 2.0 version of these widgets. To use the Motif 2.0 widgets, change the **Dt** names for the class, type, and creation routines to **Xm**. For example, change all occurrences of **DtComboBox** to **XmComboBox** in your code. This information is supplied in case you choose to port your application to Motif 2.0, but it is not a recommendation that you do so.

Note: Common Desktop Environment does not provide strict API or binary compatibility between the Common Desktop Environment widgets and the Motif 2.0 widgets.

Convenience Functions

The **DtComboBox** widget provides the following convenience functions, described in more detail in the following table.

See the **DtComboBox(3X)** man page for more information.

DtComboBox Widget Convenience Functions

Function	Description
`DtCreateComboBox()`	Creates a DtComboBox widget.
`DtComboBoxAddItem()`	Adds an item into a DtComboBox widget at a specified position.
`DtComboBoxDeletePos()`	Deletes a specified item from a DtComboBox widget.
`DtComboBoxSetItem()`	Selects an item in the XmList of a DtComboBox widget and makes it the first visible item in the list.
`DtComboBoxSelectItem()`	Selects an item in the XmList of the DtComboBox widget.

DtComboBox is a subclass of the **XmManager** class that is used to display **XmList** or **XmScrolledList**.

Classes

DtComboBox inherits behavior and resources from **Core**, **Composite**, **Constraints**, and **XmManager** classes.

The class pointer is **dtComboBoxWidgetClass**.

The class name is **DtComboBoxWidget**.

DtComboBoxWidget does not support subclassing.

Resources

DtComboBox provides the following resources. The following table shows the class, type, default, and access for these resources.

- **DtNmarginHeight** specifies the number of pixels added between the top and bottom of the text widget and the start of the shadow.
- **DtNmarginWidth** specifies the number of pixels added between the right and left sides of the text widget and the start of the shadow.
- **DtNselectedItem** is passed through to the **XmList** to set the **XmNselectedItemCount** and **XmNselectedItems** as the single item in the **DtNitems** that matches this specified **XmString** in the list.
- **DtNselectedPosition** is passed through to the **XmList** to set the **XmNselectedItemCount** and **XmNselectedItems** as the single item at this specified position in the list.
- **DtNselectionCallback** is issued when an item is selected from the **DtComboBox** widget list.
- **DtNcomboBoxType** determines the style type of the **DtComboBox**.

The list widget ID is accessible using the **XtNameToWidget()** function. The resources of these widgets can be set. See the **DtComboBox(3X)** man page for more information.

The codes in the access column show if you can:

- Set the resource at creation time (C)
- Set by using **XtSetValues** (S)
- Retrieve by using **XtGetValues** (G)

DtComboBox Widget Resources

Name	Class	Type	Default	Access
DtNmarginHeight	DtCMarginHeight	Dimension	2	CSG
DtNmarginWidth	DtCMarginWidth	Dimension	2	CSG
DtNselectedItem	DtCSelectedItem	XmString	Dimension	CSG
DtNselectedPosition	DtCSelectedPosition	int	Dimension	CSG
DtNselectionCallback	DtCCallback	XtCallbackList	XmString	C
DtNcomboBoxType	DtCComboBoxType	unsigned char	int	CG

Callback Structures

The callback structure follows and is described in the following table.

```
typedef struct {
      int        reason;
      XEvent     *event;
      XmString   item_or_text;
      int        item_position;
} DtComboBoxCallbackStruct;
```

DtComboBox Callback Structures	
Structure	Description
reason	The only reason to issue this callback is XmCR_SELECT.
event	A pointer to the event that caused this callback to be invoked. It can be NULL.
item_or_text	The contents of the text widget at the time the event invoked the callback. This data is only valid within the scope of the call_data structure, so it must be copied when it is used outside of this scope.
item_position	The new value of the DtNposition resource in the DtComboBox list. If the value is 0, the user entered a value in the XmTextField widget.

Example of DtComboBox Widget

The following example shows how to create and use a **DtComboBox** widget. You can find this code as part of the **controls.c** demo in the /usr/dt/examples/dtwidget directory.

```
/*
 * Example code for DtComboBox
 */

#include <Dt/ComboBox.h>

static char *comboValueStrings[] = {
   “alpha”, “beta”, “gamma”, “delta”,
   “epsilon”, “zeta”, “eta”, “theta”,
   “iota”, “kappa”, “lambda”, “mu”,
   “nu”, “xi”, “omicron”, “pi”,
   “rho”, “sigma”, “tau”, “upsilon”,
   “phi”, “chi”, “psi”, “omega”
};

static char *colorStrings[] = { “Red”, “Yellow”, “Green”,
“Brown”, “Blue” };

static void CreateComboBoxes(Widget parent)
{
    Widget titleLabel, comboBox, list;
    XmString *valueXmstrings, *colorXmstrings;
    int numValueStrings, numColorStrings;
    XmString labelString, xmString;
    Arg args[20];
    int i, n;

    /* Create value compound strings */

    numValueStrings = XtNumber(comboValueStrings);
    valueXmstrings = (XmString *)XtMalloc(numValueStrings *
```

```
sizeof(XmString*));
for (i = 0; i < numValueStrings; i++) {
    valueXmstrings[i] =
    XmStringCreateLocalized(comboValueStrings[i]);
}

/* Create color compound strings */
numColorStrings = XtNumber(colorStrings);
colorXmstrings = (XmString *)XtMalloc(numColorStrings *
sizeof(XmString*));

for (i = 0; i < numColorStrings; i++) {
    colorXmstrings[i] =
    XmStringCreateLocalized(colorStrings[i]);
}

/* Create title label */

labelString = XmStringCreateLocalized("ComboBox Widget");
n = 0;
XtSetArg(args[n], XmNlabelString, labelString); n++;
titleLabel = XmCreateLabel(parent, "title", args, n);
XtManageChild(titleLabel);
XmStringFree(labelString);

/*
* Create an editable ComboBox containing the color strings.
* Get the widget id of the drop down list, add some greek
* letter names to it, and make more items visible.
*/

n = 0;
XtSetArg(args[n], DtNcomboBoxType, DtDROP_DOWN_COMBO_BOX);
n++;
XtSetArg(args[n], DtNitems, colorXmstrings); n++;
XtSetArg(args[n], DtNitemCount, numColorStrings); n++;
XtSetArg(args[n], DtNvisibleItemCount, 5); n++;
XtSetArg(args[n], DtNcolumns, 10); n++;
comboBox = DtCreateComboBox(parent, "comboBox1", args, n);
XtManageChild(comboBox);

list = XtNameToWidget(comboBox, "*List");
XmListAddItems(list, valueXmstrings, 10, 0);
XtVaSetValues(list, XmNvisibleItemCount, 10, NULL);

/*
* Create an editable ComboBox with no entries.
* Get the widget id of the drop down list, add some greek
* letter names to it and select the third item in the list.
*/

n = 0;
XtSetArg(args[n], DtNcomboBoxType, DtDROP_DOWN_COMBO_BOX);
n++;
XtSetArg(args[n], DtNorientation, DtLEFT); n++;
XtSetArg(args[n], DtNcolumns, 10); n++;
comboBox = DtCreateComboBox(parent, "comboBox2", args, n);
XtManageChild(comboBox);
```

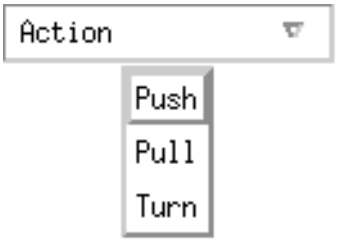
```
    list = XtNameToWidget(comboBox, "*List");
    XmListAddItems(list, valueXmstrings, 7, 0);
    XtVaSetValues(list, XmNvisibleItemCount, 7, NULL);
    XtVaSetValues(comboBox, DtNselectedPosition, 3, NULL);

    /*
     * Create a non-editable ComboBox containing some greek
              letter names.
     * Position the arrow on the left.
     * Select the 'gamma' item in the list.
     */

    n = 0;
    XtSetArg(args[n], DtNorientation, DtLEFT); n++;
    XtSetArg(args[n], DtNitems, valueXmstrings); n++;
    XtSetArg(args[n], DtNitemCount, numValueStrings); n++;
    XtSetArg(args[n], DtNvisibleItemCount, 8); n++;
    comboBox = DtCreateComboBox(parent, "comboBox3", args, n);
    XtManageChild(comboBox);

    xmString = XmStringCreateLocalized("gamma");
    XtVaSetValues(comboBox, DtNselectedItem, xmString, NULL);
    XmStringFree(xmString);

    /*
     * Create a non-editable ComboBox with no entries.
     * Position the arrow on the right.
     * Add the greek letter names to the list and select the
              fourth item.
     */

    n = 0;
    XtSetArg(args[n], DtNorientation, DtRIGHT); n++;
    XtSetArg(args[n], DtNvisibleItemCount, 8); n++;
    comboBox = DtCreateComboBox(parent, "comboBox4", args, n);
    XtManageChild(comboBox);

    for (i = 0; i < numValueStrings; i++) {
        DtComboBoxAddItem(comboBox, valueXmstrings[i],
        0, True);
    }
    XtVaSetValues(comboBox, DtNselectedPosition, 4, NULL);
    /*
     * Free value and color strings, ComboBox has taken a copy.
     */

    for (i = 0; i < numValueStrings; i++) {
        XmStringFree(valueXmstrings[i]);
    }
    XtFree((char*)valueXmstrings);

for (i = 0; i < numColorStrings; i++) {
       XmStringFree(colorXmstrings[i]);
```

```
        }
        XtFree((char*)colorXmstrings);
    }
```

Menu Button Widget (DtMenuButton)

Use the **DtMenuButton** widget to provide menu-cascading functionality outside of a menu pane.

DtMenuButton widget is a command widget that complements the menu cascading functionality of an **XmCascadeButton** widget. As a complement to XmCascadeButton widget, it can only be instantiated outside a MenuBar, Pulldown, or Popup (use **XmCascadeButton** widget inside a MenuPane.) The following figure shows examples of a **DtMenuButton** widget.



Examples of a DtMenuButton Widget

Library and Header Files

The **DtMenuButton** widget is in the **libDtWidget** library. The header file is **Dt/MenuButton.h**.

Demo Program

A demo containing an example of the **DtMenuButton** widget is in **/usr/dt/examples/dtwidget/controls.c**.

Convenience Functions

DtCreateMenuButton() is a convenience function that creates a Common Desktop Environment widget.

DtMenuButton widget is a subclass of **XmLabel** class. Visually, **DtMenuButton** widget has a label string and a menu glyph. The menu glyph always appears on the right end of the widget and, by default, is a downward pointing arrow.

DtMenuButton widget has an implicitly created submenu attached to it. The submenu is a pop-up menu and has this **DtMenuButton** widget as its parent. The name of the implicitly created submenu is obtained by prefixing submenu_ to the name of this **DtMenuButton** widget. You can obtain the widget ID of the submenu by setting an **XtGetValues** on **DtNsubMenuId** resource of this **DtMenuButton** widget. The implicitly created submenu must not be destroyed by the user of this widget.

The submenu can be popped up by pressing the menu post button (see **XmNmenuPost** resource of **XmRowColumn**) anywhere on the DtMenuButton widget or by pressing the Motif Cancel key (usually Escape).

Classes

DtMenuButtonWidget inherits behavior and resources from **Core**, **XmPrimitive**, and **XmLabel** classes.

The class pointer is **dtMenuButtonWidgetClass**.

The class name is **DtMenuButtonWidget**.

DtMenuButtonWidget does not support subclassing.

Resources

DtMenuButtonWidget provides the following resources. The following table shows the class, type, default, and access for these resources.

- **DtNcascadingCallback** specifies the list of callbacks that are called before the attached submenu is displayed.
- **DtNcascadePixmap** specifies the pixmap that is displayed as the menu glyph. If no pixmap is specified, a downward pointing arrow is displayed.
- **DtNsubMenuId** specifies the widget ID of the pop-up menu pane to be associated with this **DtMenuButton** widget. You must create the pop-up menu pane with this **DtMenuButton** as its parent. You cannot specify this resource when the widget is created because the submenu is automatically destroyed by this widget when the resource is set.

See the **DtMenuButtonWidget(3X)** man page for more information.

The codes in the access column show if you can:

- Set the resource at creation time (C)
- Set by using **XtSetValues** (S)
- Retrieve by using **XtGetValues** (G)

DtMenuButtonWidget Resources

Name	Class	Type	Default	Access
`DtNcascadingCallback`	`DtCCallback`	`XtCallbackList`	`NULL`	`C`
`DtNcascadePixmap`	`DtCPixmap`	`Pixmap`	`XmUNSPECIFIED_PIXMAP`	`CSG`
`DtNsubMenuId`	`DtCMenuWidget`	`Widget`	`NULL`	`SG`

Callback Structures

The callback structure follows and is described in the following table.

```
typedef struct {
    int        reason;
     XEvent     *event;
} XmAnyCallbackStruct;
```

DtMenuButtonWidget Callback Structures

Structure	Description
reason	Returns reason why the callback was invoked.
event	Points to the XEvent that triggered the callback or NULL if the callback was not triggered by an XEvent.

Example of DtMenuButton Widget

The following example shows how to create and use a DtMenuButton widget. You can find this code as part of the **controls.c** demo in the /usr/dt/examples/dtwidget directory.

```
/*
 * Example code for DtMenuButton
 */

#include Dt/DtMenuButton.h

/* MenuButton custom glyph */

#define menu_glyph_width 16
#define menu_glyph_height 16
static unsigned char menu_glyph_bits[] = {
    0xe0, 0x03, 0x98, 0x0f, 0x84, 0x1f, 0x82, 0x3f, 0x82, 0x3f,
    0x81,
    0x7f,
    0x81, 0x7f, 0xff, 0x7f, 0xff, 0x40, 0xff, 0x40, 0xfe, 0x20,
    0xfe,
    0x20,
    0xfc, 0x10, 0xf8, 0x0c, 0xe0, 0x03, 0x00, 0x00};

static void CreateMenuButtons(Widget parent)
{
    Widget menuButton, submenu, titleLabel, button;
    Pixmap cascadePixmap;
    Pixel fg, bg;
    Cardinal depth;
    XmString labelString;
    Arg args[20];
    int i, n;

    /* Create title label */

    labelString = XmStringCreateLocalized(“MenuButton Widget”);
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    titleLabel = XmCreateLabel(parent, “title”, args, n);
    XtManageChild(titleLabel);
    XmStringFree(labelString);

    /*
```

```
 * Create a MenuButton.
 * Add push buttons to the built-in popup menu.
 */

labelString = XmStringCreateLocalized("Action");
n = 0;
XtSetArg(args[n], XmNlabelString, labelString); n++;
menuButton = DtCreateMenuButton(parent, "menuButton1", args,
n);
XtManageChild(menuButton);
XmStringFree(labelString);

XtVaGetValues(menuButton, DtNsubMenuId, &submenu, NULL);
button = XmCreatePushButton(submenu, "Push", NULL, 0);
XtManageChild(button);
button = XmCreatePushButton(submenu, "Pull", NULL, 0);
XtManageChild(button);
button = XmCreatePushButton(submenu, "Turn", NULL, 0);
XtManageChild(button);

/*
 * Create a MenuButton.
 * Replace the built-in popup menu with a tear-off menu.
 * Add a custom pixmap in the colors of the MenuButton.
 */

labelString = XmStringCreateLocalized("Movement");
n = 0;
XtSetArg(args[n], XmNlabelString, labelString); n++;
menuButton = DtCreateMenuButton(parent, "menuButton1", args,
n);
XtManageChild(menuButton);
XmStringFree(labelString);

/* Create a tear-off menu */

n = 0;
XtSetArg(args[0], XmNtearOffModel, XmTEAR_OFF_ENABLED); n++;
submenu = XmCreatePopupMenu(menuButton, "submenu", args, n);
button = XmCreatePushButton(submenu, "Run", NULL, 0);
XtManageChild(button);
```

```
    button = XmCreatePushButton(submenu, “Jump”, NULL, 0);
    XtManageChild(button);
    button = XmCreatePushButton(submenu, “Stop”, NULL, 0);
    XtManageChild(button);

    XtVaSetValues(menuButton, DtNsubMenuId, submenu, NULL);

    /* Create a pixmap using the menu button’s colors and depth
*/

    XtVaGetValues(menuButton,
                  XmNforeground, &fg,
                  XmNbackground, &bg,
                  XmNdepth, &depth,
                  NULL);
    cascadePixmap = XCreatePixmapFromBitmapData(XtDisplay
                  (menuButton),DefaultRootWindow(XtDisplay
                  (menuButton)),
                  (char*)menu_glyph_bits,
                  menu_glyph_width, menu_glyph_height,
                  fg, bg, depth);
    XtVaSetValues(menuButton, DtNcascadePixmap, cascadePixmap,
    NULL);
}
```

Text Editor Widget (DtEditor)

The Common Desktop Environment text editing system consists of two components:

- The text editor client, **dtpad**, which provides editing services through graphical, action, and ToolTalk interfaces.
- The editor widget, **DtEditor(3)**, which provides a programmatic interface for the following editing services:
- Cut and paste
- Search and replace
- Simple formatting
- Spell checking (for 8-bit locales)
- Undo previous edit
- Enhanced I/O handling capabilities that support input and output of ASCII text, multibyte text, and buffers of data
- Support for reading and writing files directly

Although the OSF/Motif text widget also provides a programmatic interface, applications that use the system-wide uniform editor should use the **DtEditor(3)** widget. The Common

Desktop Environment Text Editor and Mailer use the editor widget. Use this widget in the following circumstances:

1.. You want the functionality, such as spell checking, undo, and find/change, that is provided by the **DtEditor(3)** widget.

2.. You do not want to write the code so that users may read and write data to and from a file.

3.. Your program does not need to examine every character typed or every cursor movement made by a user.

This section describes the text editor widget, **DtEditor(3)**.

The Editor Widget library provides support for creating and editing text files. It enables applications running in the desktop environment to have a consistent method of editing text data. The **DtEditor(3)** widget consists of a scrolled edit window for text, an optional status line, and dialogs for finding and changing text, spell checking, and specifying formatting options. The text editor widget includes a set of convenience functions for programmatically controlling the widget.

Library and Header Files

The **DtEditor** widget is in the libDtWidget library. The header file is Dt/Editor.h.

Demo Program

A demo containing an example of the **DtEditor** widget is in **/usr/dt/examples/dtwidget/editor.c**.

Classes

Widget subclassing is not supported for the DtEditor widget class.

DtEditor inherits behavior and resources from **Core**, **Composite**, **Constraints**, **XmManager**, and **XmForm** classes.

The class name for the editor widget is **DtEditorWidget**.

The class pointer is **dtEditorWidgetClass**.

Convenience Functions

The **DtEditor** convenience functions are described in the following tables.

Life Cycle Functions

The **DtEditor** life cycle functions are described in the following table.

DtEditor Life Cycle Functions

Function	Description
`DtCreateEditor`	Creates a new instance of a DtEditor **widget and its children.**
`DtEditorReset`	Restores a DtEditor widget to its initial state.

Input/Output Functions

The **DtEditor** input/output functions are described in the following table.

DtEditor Input/Output Functions	
Function	Description
`DtEditorAppend`	Appends content data to the end of an editor widget.
`DtEditorAppendFromFile`	Appends the contents of a file to the end of an editor widget.
`DtEditorGetContents`	Retrieves the entire contents of an editor widget.
`DtEditorInsert`	Inserts content data at the current insert position.
`DtEditorInsertFromFile`	Inserts the contents of a file at the current insert position.
`DtEditorReplace`	Replaces a portion of text with the supplied data.
`DtEditorReplaceFromFile`	Replaces a portion of text with the contents of a file.
`DtEditorSaveContentsToFile`	Saves the entire contents to a file.
`DtEditorSetContents`	Loads content data into an editor widget, replacing the entire contents of the widget.
`DtEditorSetContentsFromFile`	Loads the contents of a file into an editor widget, replacing the entire contents of the widget.

Selection Functions

The **DtEditor** selection functions are described in the following table.

DtEditor Selection Functions	
Function	Description
`DtEditorClearSelection`	Replaces the currently selected contents with blanks.
`DtEditorCopyToClipboard`	Copies the currently selected contents to the clipboard.
`DtEditorCutToClipboard`	Removes the currently selected contents, placing then on the clipboard.
`DtEditorDeleteSelection`	Removes the currently selected contents.
`DtEditorDeselect`	Deselects any selected contents.
`DtEditorPasteFromClipboard`	Pastes the contents of the clipboard into an editor widget, replacing any currently selected contents.
`DtEditorSelectAll`	Selects the entire contents of an editor widget.

Format Functions

The **DtEditor** format functions are described in the following table.

DtEditor Format Functions	
Function	Description
`DtEditorFormat`	Formats all or part of the contents of an editor widget.
`DtEditorInvokeFormatDialog`	Displays the format dialog box that enables the user to specify format settings for margins and justification style and to perform formatting operations.

Find and Change Functions

The **DtEditor** find and change functions are described in the following table.

DtEditArea Find and Change Functions	
Function	Description
`DtEditorChange`	Changes one or all occurrences of a string.
`DtEditorFind`	Finds the next occurrence of a string.
`DtEditorInvokeFindChangeDialog`	Displays the dialog box that enables the user to search for, and optionally change, a string.
`DtEditorInvokeSpellDialog`	Displays a dialog box with a list of misspelled words in the current contents.

Auxiliary Functions

The **DtEditor** auxiliary functions are described in the following table.

DtEditor Auxiliary Functions	
Function	Description
`DtEditorCheckForUnsavedChanges`	Reports whether the contents of an editor widget have been altered since the last time they were retrieved or saved.
`DtEditorDisableRedisplay`	Prevents redisplay of an editor widget even though its visual attributes have changed.
`DtEditorEnableRedisplay`	Forces the visual update of an editor widget.
`DtEditorGetInsertPosition`	Returns the insertion cursor position of the editor widget.
`DtEditorGetLastPosition`	Returns the position of the last character in the edit window.

DtEditor Auxiliary Functions	
Function	Description
`DtEditorGetMessageTextFieldID`	Retrieves the widget ID of the text field widget used to display application messages.
`DtEditorGetSizeHints`	Retrieves sizing information from an editor widget.
`DtEditorGoToLine`	Moves the insertion cursor to the specified line.
`DtEditorSetInsertionPosition`	Sets the position of the insertion cursor.
`DtEditorTraverseToEditor`	Sets keyboard traversal to the edit window of an editor widget.
`DtEditorUndoEdit`	Undoes the last edit made by a user.

Resources

The **DtEditor** widget provides the following set of resources.

- **DtNautoShowCursorPosition** ensures that the text visible in the scrolled edit window contains the insert cursor when set to True. If the insert cursor changes, the contents of the editor may scroll to bring the insertion point into the window.
- **DtNblinkRate** specifies the blink rate of the text cursor in milliseconds. The time it takes to blink the insertion cursor on and off is twice the blink rate. When the blink rate is set to zero, the cursor does not blink. The value must not be negative.
- **DtNbuttonFontList** specifies the font list used for the buttons that are displayed in the dialog boxes of **DtEditor**.
- **DtNcolumns** specifies the initial width of the editor as an integer number of characters. The value must be greater than zero.
- **DtNcursorPosition** specifies the location of the current insert cursor in the editor where the current insert cursor is placed. Position is determined by the number of characters from the beginning of the text. The first character position is 0.
- **DtNcursorPositionVisible** marks the insert cursor position by a blinking text cursor when the Boolean value is True.
- **DtNdialogTitle** specifies the title for all dialogs displayed by DtEditor. These include the dialogs for word search and replace, misspelled words, and format settings.
- **DtNeditable** indicates that the user can edit the data when set to True. Prohibits the user from editing data when set to False.
- **DtNlabelFontList** specifies the font list used for **DtEditor** labels (the labels are displayed in the status line and **DtEditor** dialog boxes).
- **DtNoverstrike** when set to False, characters typed into the editor widget inserts at the position of the cursor (the default). When set to True, characters typed into the editor widget replace the characters that directly follow the insertion cursor. When the end of the line is reached, characters are appended to the end of the line. If the status line is visible, the **DtNoverstrikeIndicatorLabel** is displayed in the status line whenever **DtNoverstrike** is True.

- **DtNrows** specifies the initial height of the editor measured in character heights. The value must be greater than zero.
- **DtNscrollHorizontal** adds a scroll bar that enables the user to scroll horizontally through text when the Boolean value is True.
- **DtNscrollLeftSide** puts a vertical scroll bar on the left side of the scrolled edit window when the Boolean value is True.
- **DtNshowStatusLine** displays a status line below the text window when set to True. The status line contains a field that displays the current line number of the insert cursor, total number of lines in the document, and whether the editor is in overstrike mode. Users can type a line number in the line number display to go directly to that line.

The status line also includes a Motif **XmTextField(3X)** widget for displaying messages supplied by an application. This field is a convenient place for an application to display status and feedback about the document being edited. The ID of the text field is retrieved using **DtEditorGetMessageTextFieldID(3)**. A message is displayed by setting the **XmNvalue** or **XmNvalueWcs** resource of this widget. If the text field is not needed, you can unmanage it by calling **XtUnmanageWidget(3X)** with its ID.

- **DtNspellFilter** specifies the filter used to identify spelling errors. The function **DtEditorInvokeSpellDialog(3)** filters the contents of an editor through the filter specified by **DtNspellFilter**. The filter specified should accept a file name and produce a list of misspelled and unrecognized words in this file on **stdout**. The default filter is **spell(1)**.
- **DtNtextBackground** specifies the background for the edit window.
- **DtNtextDeselectCallback** specifies a function called whenever no text is selected within the edit area. The reason sent by the callback is **DtEDITOR_TEXT_DESELECT**.
- **DtNtextFontList** specifies the font list used for the **DtEditor** edit window and its text fields. The text fields are displayed in the status line and **DtEditor** dialog boxes.
- **DtNtextForeground** specifies the foreground for the edit window.
- **DtNtextSelectCallback** specifies a function called whenever text is selected within the edit area. The reason sent by the callback is **DtEDITOR_TEXT_SELECT**.
- **DtNtextTranslations** specifies translations that are added to the edit window. Translations specified with this resource override any duplicate translations defined for the edit window. See the **DtEditor(3)** man page for a list of translations provided by **DtEditor**.
- **DtNtopCharacter** displays the line that contains the position of text at the top of the scrolled edit window. The line is displayed at the top of the widget without shifting the text left or right. Position is determined by the number of characters from the beginning of the text. The first character position is zero.

XGetValues(3X) for **DtNtopCharacter** returns the position of the first character in the line that is displayed at the top of the widget.

- **DtNwordWrap** breaks lines at word breaks with soft carriage returns when they reach the right edge of the window. Note that word wrap affects only the visual appearance of the contents of an editor widget. The line breaks (soft carriage returns) are not physically inserted into the text. The editor does support substituting hard carriage returns when the contents of the widget are retrieved or saved to a file. See the **DtEditorGetContents(3)** and **DtEditorSaveContentsToFile(3)** man pages for more information.

The following table lists the class, type, default, and access for each resource. You can also set the resource values for the inherited classes to set attributes for this widget. To reference a resource by name or class in an .**Xdefaults** file, remove the **DtN** or **DtC** prefix

and use the remaining letters. To specify one of the defined values for a resource in an **.Xdefaults** file, remove the Dt prefix and use the remaining letters (in either lowercase or uppercase, but include any underscores between words).

The codes in the access column show if you can:

- Set the resource at creation time (C)
- Set by using **XtSetValues** (S)
- Retrieve by using **XtGetValues** (G)

See the **DtEditor(3)** man page for more information.

DtEditor Resources

Name	Class	Type	Default	Access
DtNautoShowCursorPosition	DtCAutoShowCursorPosition	Boolean	True	CSG
DtNblinkRate	DtCBlinkRate	int	500	CSG
DtNbuttonFontList	DtCFontList	XmFontList	Dynamic	CSG
DtNcolumns	DtCColumns	XmNcolumns	Dynamic	CSG
DtNcursorPosition	DtCCursorPosition	XmTextPosition	0	CSG
DtNcursorPositionVisible	DtCCursorPositionVisible	Boolean	True	CSG
DtNdialogTitle	DtCDialogTitle	XmString	NULL	CSG
DtNeditable	DtCEditable	Boolean	True	CSG
DtNlabelFontList	DtCFontList	XmFontList	Dynamic	CSG
DtNmaxLength	DtCMaxLength	int	Largest integer	CSG
DtNoverstrike	DtCOverstrike	Boolean	False	CSG
DtNrows	DtCRows	XmNrows	Dynamic	CSG
DtNscrollHorizontal	DtCScroll	Boolean	True	CG
DtNscrollLeftSide	DtCScrollSide	Boolean	Dynamic	CG
DtNscrollTopSide	DtCScrollSide	Boolean	False	CG
DtNscrollVertical	DtCScroll	Boolean	True	CG
DtNshowStatusLine	DtCShowStatusLine	Boolean	False	CSG
DtNspellFilter	DtCspellFilter	char *	Spell	CSG
DtNtextBackground	DtCBackground	Pixel	Dynamic	CSG
DtNtextDeselectCallback	DtCCallback	XtCallbackList	NULL	C
DtNtextFontList	DtCFontList	XmFontList	Dynamic	CSG
DtNtextForeground	DtCForeground	Pixel	Dynamic	CSG
DtNtextTranslations	DtCTranslations	XtTranslations	NULL	CS

DtEditor Resources				
Name	Class	Type	Default	Access
DtNtextSelectCallback	DtCCallback	XtCallbackList	NULL	C
DtNtopCharacter	DtCTextPosition	XmTextPosition	0	CSG
DtNwordWrap	DtCWordWrap	Boolean	True	CSG

Inherited Resources

DtEditor inherits behavior and resources from the following superclasses:

- **XmForm**
- **XmManager**
- **Composite**
- **Core**

Refer to the appropriate man page for more information.

Localization Resources

The following list describes a set of widget resources that are designed for localization of the **DtEditor** widget and its dialog boxes. Default values for these resources depend on the locale.

- **DtNcenterToggleLabel** specifies the label for the center alignment toggle button in the Format Settings dialog box. The default value in the C locale is Center.
- **DtNchangeAllButtonLabel** specifies the label for the button in the Find/Change dialog box that changes all occurrences of the Find string in the document. The default value in the C locale is Change All.
- **DtNchangeButtonLabel** specifies the label for the button in the Find/Change dialog box that changes the next occurrence of the find string in the document. The default value in the C locale is Change.
- **DtNchangeFieldLabel** specifies the label for the field in the Find/Change dialog box where the user specifies the replacement string. The default value in the C locale is Change To.
- **DtNcurrentLineLabel** specifies the label for the current line number field in the status line. The default value in the C locale is Line.
- **DtNfindButtonLabel** specifies the label for the button in the Find/Change dialog box that finds the next occurrence of the find string in the document. The default value in the C locale is Find.
- **DtNfindChangeDialogTitle** specifies the title for the Find/Change dialog box. If **DtNdialogTitle** is non-null, it is added to the front of this resource to form the title. The default value in the C locale is Find/Change.
- **DtNfindFieldLabel** specifies the label for the field in the Find/Change dialog box where the user specifies the search string. The default value in the C locale is Find.
- **DtNformatAllButtonLabel** specifies the label for the button in the Format Settings dialog box that formats the complete document. The default value in the C locale is All.

- **DtNformatParagraphButtonLabel** specifies the label for the button in the Format Settings dialog box that formats the paragraph containing the insertion cursor. The default value in the C locale is Paragraph.
- **DtNformatSettingsDialogTitle** specifies the title for the Format Settings dialog box. If DtNdialogTitle is non-null, it is added to the front of this resource to form the title. The default value in the C locale is Format Settings.
- **DtNinformationDialogTitle** specifies the title for the Information dialog box that is used to present feedback and general information to the user. If **DtNdialogTitle** is non-null, it is added to the front of this resource to form the title. The default value in the C locale is Information.
- **DtNjustifyToggleLabel** specifies the label for the justify alignment toggle button in the Format Settings dialog box. The default value in the C locale is Justify.
- **DtNleftAlignToggleLabel** specifies the label for the left alignment toggle button in the Format Settings dialog box. The default value in the C locale is Left Align.
- **DtNleftMarginFieldLabel** specifies the label for the left margin value field in the Format Settings dialog box. The default value in the C locale is Left Margin.
- **DtNmisspelledListLabel** specifies the label for the list of unrecognized and misspelled words in the Spell dialog box. The default value in the C locale is Misspelled Words.
- **DtNoverstrikeLabel** specifies the label in the status line that shows that the editor is in overstrike mode. The default value in the C locale is Overstrike.
- **DtNrightAlignToggleLabel** specifies the label for the right alignment toggle button in the Format Settings dialog box. The default value in the C locale is Right Align.
- **DtNrightMarginFieldLabel** specifies the label for the right margin value field in the Format Settings dialog box. The default value in the C locale is Right Margin.
- **DtNspellDialogTitle** specifies the title for the Format Settings dialog box. If **DtNdialogTitle** is non-null, it is added to the front of this resource to form the title. The default value in the C locale is Spell.
- **DtNtotalLineCountLabel** specifies the label for the display as part of the status line that shows the total number of lines in the document. The default value in the C locale is Total.

The following table lists the class, type, default, and access for each of the localization resources. The codes in the access column show if you can:

- Set the resource at creation time (C)
- Set by using **XtSetValues** (S)
- Retrieve by using **XtGetValues** (G)

See the **DtEditor(3)** man page for more information.

DtEditor Localization Resources

Name	Class	Type	Default	Access
`DtNcenterToggleLabel`	`DtCCenterToggleLabel`	`XmString`	`Dynamic`	`CSG`
`DtNchangeAllButtonLabel`	`DtCChangeAllButtonLabel`	`XmString`	`Dynamic`	`CSG`
`DtNchangeButtonLabel`	`DtCChangeButtonLabel`	`XmString`	`Dynamic`	`CSG`

DtEditor Localization Resources				
Name	Class	Type	Default	Access
DtNchangeFieldLabel	DtCChangeFieldLabel	XmString	Dynamic	CSG
DtNcurrentLineLabel	DtCCurrentLineLabel	XmString	Dynamic	CSG
DtNfindButtonLabel	DtCFindButtonLabel	XmString	Dynamic	CSG
DtNfindChangeDialogTitle	DtCFindChangeDialogTitle	XmString	Dynamic	CSG
DtNfindFieldLabel	DtCFindFieldLabel	XmString	Dynamic	CSG
DtNformatAllButtonLabel	DtCFormatAllButtonLabel	XmString	Dynamic	CSG
DtNformatParagraphButtonLabel	DtCFormatParagraphButtonLabel	XmString	Dynamic	CSG
DtNformatSettingsDialogTitle	DtCFormatSettingsDialogTitle	XmString	Dynamic	CSG
DtNinformationDialogTitle	DtCInformationDialogTitle	XmString	Dynamic	CSG
DtNjustifyToggleLabel	DtCJustifyToggleLabel	XmString	Dynamic	CSG
DtNleftAlignToggleLabel	DtCLeftAlignToggleLabel	XmString	Dynamic	CSG
DtNleftMarginFieldLabel	DtCLeftMarginFieldLabel	XmString	Dynamic	CSG
DtNmisspelledListLabel	DtCMisspelledListLabel	XmString	Dynamic	CSG
DtNoverstrikeLabel	DtCOverstrikeLabel	XmString	Dynamic	CSG
DtNrightAlignToggleLabel	DtCRightAlignToggleLabel	XmString	Dynamic	CSG
DtNrightMarginFieldLabel	DtCRightMarginFieldLabel	XmString	Dynamic	CSG
DtNspellDialogTitle	DtCSpellDialogTitle	XmString	Dynamic	CSG
DtNtotalLineCountLabel	DtCTotalLineCountLabel	XmString	Dynamic	CSG

Callback Functions

The DtEditor widget supports three callback functions:

- `DtEditorNHelpCallback`
- `DtNtextSelectCallback`
- `DtNtextDeselectCallback`

If you want to present help information about the editor widget and its dialog boxes, set the **XmNhelpCallback** resource and use the reason field passed as part of **DtEditorHelpCallbackStruct** to set the contents of the Help dialog box. A pointer to the following structure is passed to **XmNHelpCallback**. The callback structure and is described in the following table.

```
typedef struct {
    int       reason;
    XEvent    *event;
} DtEditorHelpCallbackStruct;
```

DtEditorHelp Callback Structure	
Structure	Description
reason	The reason why the callback was invoked. Refer to the DtEditor(3) man page for a list of reasons.
event	A pointer to the event that invoked this callback. The value can be NULL.

Use the **DtNtextSelectCallback** and **DtNtextDeselectCallback** resources when you want to enable and disable menu items and commands depending on whether text is selected. **DtNtextSelectCallback** specifies a function that is called whenever some text is selected in the edit window. **DtNtextDeselectCallback** specifies a function that is called whenever no text is selected within the edit window. The reasons sent by the callbacks are DtEDITOR_TEXT_SELECT and DtEDITOR_TEXT_DESELECT.

Invoking Actions from Applications

If your application manages an extensible collection of data types, there is a strong likelihood that it should be directly involved with action invocation. This section explains how you can invoke an action from an application. Included is an example program that shows you how to invoke an action.

For more information on actions and how you create them, see the section “Accessing the Data–Typing Database” in this manual, and the following sections in the *Advanced User’s and System Administrator’s Guide*:

- “Introduction to Actions and Data Types”
- “Creating Actions and Data Types Using Create Action”
- “Creating Actions Manually”
- “Creating Data Types Manually”

This section contains the following sections:

- Mechanisms for Invoking Actions from an Application
- Types of Actions
- Action Invocation API
- Related Information
- actions.c Example Program
- Listing for actions.c

Mechanisms for Invoking Actions from an Application

The action invocation API exported by the Desktop Services library is one mechanism available to your application to cause another application to be invoked or to perform an operation. Other mechanisms include:

- The **fork/exec** system calls
- ToolTalk messages

  Each of these mechanisms has benefits and limitations, so you must evaluate your specific situation to determine which is most appropriate.

  The advantages of using the action invocation API include:

- Actions can encapsulate both traditional command–line applications (that is, **COMMAND** actions) and ToolTalk applications (that is, **TT_MSG** actions). The application that invokes the action does not need to know whether a command is forked or a message is sent.
- Actions are polymorphic and are integrated with the desktop’s data–typing mechanisms. This means that an action, such as Open or Print, may have different behavior depending on the type of argument that is supplied, but the behavior differences are transparent to the application that invokes the action.
- Actions provide a great deal of configurability for the application developer, system integrator, system administrator, and end user. Any one of these people can edit the action database to modify the definition of how an action is to be performed.
- Actions work well in distributed environments. If an application uses **fork/exec** to directly invoke another application, then both applications must be available and able to run on

the same system. By contrast, the action invocation API uses information in the action database to determine on which system a **COMMAND** action should be invoked.

- Actions allow your application to behave consistently with the behavior of the desktop. This is because the desktop's components interact by using actions when manipulating the user's data files.
- The disadvantage of using g the action invocation API is that it is only an invocatin mechanism that has limited return value capabilities and has no capabilities for a dialog with the invoked action handler. If these features are required, **fork/exec/pipes** can be used. However, within CDE, ToolTalk is the preferred cross process communications mechanism due to its generalized client/server paradigm.

Returning to invocation, suppose your application manages data files in several different formats (text and graphics) and needs to provide a way for the user to edit and display these files. To implement this without using actions, you would probably use one of the following mechanisms:

- Use **fork/exec** to start the appropriate editor and invent some mechanism (for example, environment variables) for the user to specify the names of the editors. The limitations of this approach include the following:
- You must write complex code that uses system calls to invoke sub–processes and monitors the resulting signals.
- The editors must either be available on the same system as your application or the system administrator must provide a complex configuration using facilities such as **rsh**.
- System administrators and users must learn and manage your application's unique configuration model.
- Use ToolTalk messages to request that operations, such as Edit and Display, be performed on the data. The limitation of this approach is that ToolTalk–based editors must be available for all of your types of data.

To implement this using actions, you only have to invoke the Open action on the buffer or on the data file. The action invocation API will use the action database to determine the appropriate message to send or command to invoke, as well as handle all details, such as creating and cleaning up temporary files and catching necessary signals.

Types of Actions

The action application program interface (API) works with any type of action. Types of actions in the desktop include:

Command actions
: Specifies a command line to execute.

ToolTalk actions
: Specifies a ToolTalk message to send, which is then received by the appropriate application.

Map actions
: Refers to another action instead of defining any specific behavior.

See "Introduction to Actions and Data Types" in the Common Desktop Environment: Advanced Users' and System Administrator's Guide for more information.

Action Invocation API

The action invocation API is exported from the Desktop Services library and provides functions to accomplish a number of tasks, such as:

- Initialize and load the database of action and data–type definitions. The database **must** be loaded before an action can be run.

- Query the database. There are functions to determine whether a specified action or its associated icon image, label, or description exists.
- Invoke an action. The application can pass file or buffer arguments to the action.
- Register a callback to receive action status and return arguments.

Related Information

For detailed information about action commands, functions, and data formats, see the following man pages:

- **dtaction(1)**
- **dtactionfile(4)**
- **DtActionCallbackProc(3)**
- **DtActionDescription(3)**
- **DtActionExists(3)**
- **DtActionIcon(3)**
- **DtActionInvoke(3)**
- **DtActionLabel(3)**
- **DtActionQuit(3)**
- **DtActionQuitType(3)**
- **DtActionStUpCb(3)**
- **dtexec(1)**

actions.c Example Program

This section describes a simple example program, **actions.c**. A complete listing of **actions.c** is at the end of this section.

Loading the Database of Actions and Data Types

Before your application can invoke an action, it must initialize the Desktop Services library (which contains the action invocation API) and load the database of action and data–type definitions.

To Initialize the Desktop Services Library

- Use the **DtInitialize()** function to initialize the Desktop Services Library.

```
DtInitialize(*display,widget,*name,*tool_class)
```

DtInitialize() uses the default Intrinsic **XtAppContext**. The API provides an additional function, **DtAppInitialize()** to use when your application must specify an app_context:

```
DtAppInitialize(app_context,*display,widget,*name, tool_class)
```

DtInitialize() Example

The following code segment shows how the example program **actions.c** uses **DtInitialize()**.

```
/* Initialize the desktop */
if (DtInitialize(XtDisplay(shell), shell, argv[0],
ApplicationClass)==False) {
        /* DtInitialize() has already logged an appropriate error
msg */
        exit(1);
    }
```

To Load the Actions and Data–Types Database

- Use the DtDbLoad() function to load the actions and data–typing databases.

```
DtDbLoad(void)
```

DtDbLoad() reads in the action and data–typing databases. This function determines the set of directories that are to be searched for database files (the database search path) and loads the ***.dt** files found into the database. The directory search path is based on the value of the **DTDATABASESEARCHPATH** environment variable and internal defaults.

DtDbLoad() Example

The following code segment shows how the example program **actions.c** uses **DtDbLoad()**.

```
/* Load the filetype/action databases */
    DtDbLoad();
```

To Request Notification of Reload Events

If you use **DtDbLoad()** in a long–lived application, it must dynamically reload the database whenever it is modified.

1.. Use the DtDbReloadNotify()function to request notification of reload events.

```
DtDbReloadNotify(DtDbReloadCallbackProc callback_proc,
                 XtPointer client_data)
```

2.. Supply a callback that:

- Destroys cached database information held by the application
- Calls the DtDbLoad() function again

Callback_proc cleans up any cached database information your application is holding and then invokes DtDbLoad(). Client_data may be used to pass additional client information to the callback routine.

Checking the Actions Database

Your application accesses the database if it needs to display the icon or label for an action. Also by invoking an action, your application can check that it exists. An action is identified in the database by the action name:

```
ACTION action_name
{
  o
}
```

For example, the action definition for the Calculator looks like this:

```
ACTION Dtcalc
{
        LABEL           Calculator
        ICON            Dtcalc
        ARG_COUNT       0
        TYPE            COMMAND
        WINDOW_TYPE     NO_STDIO
        EXEC_STRING     /usr/dt/bin/dtcalc
        DESCRIPTION     The Calculator (Dtcalc) action runs the \
                        desktop Calculator application.
}
```

The action name for the Calculator action is Dtcalc. When an executable file has a file name that matches an action name in the database, that file is an action file—a representation for the underlying action. The information about the icon and label for that file are stored in the database.

To Determine Whether a Specified Action Definition Exists

- Use the DtActionExists() function to determine whether a specified action definition exists.

  ```
  DtActionExists(*name)
  ```

  DtActionExists() checks whether the specified name corresponds to the name of an action in the database. The function returns True if name corresponds to an action name, or False if no action with that name is found.

To Obtain the Icon Image Information for a Specified Action

- Use the DtActionIcon() function to obtain the icon image information.

  ```
  DtActionIcon(char *action_name)
  ```

An action definition specifies the icon image used to represent the action in the definition's ICON field:

```
ACTION action_name
{
   ICON  icon_image_base_name
   o
}
```

DtActionIcon() returns a character string containing the value of the icon image field. If the action definition does not contain an icon field, the function returns the value of the default action icon image, Dtactn.

You then need to determine the location of the icon, and the size you want to use. Icons can exist in four sizes and are available in bitmap or pixmap form. For example, you can find the base name of the icon file from the action definition for the Calculator. You then use the base name coupled with the information given in the following table and knowledge of the location of all the icons to find the specific icon file you want.

The icon name for the calculator action is Dtcalc, but that is not the entire file name. Icon file names are based on the size of the icon, and there are four sizes. The following table shows the sizes and file–naming conventions for the desktop icons.

Icon Sizes and File Names		
Icon Size	Bitmap Name	Pixmap Name
16 by 16 (tiny)	name.t.bm	`name.t.pm`
24 by 24 (small)	name.s.bm	name.s.pm
32 by 32 (medium)	name.m.bm	name.m.pm
48 by 48 (large)	name.l.bm	name.l.pm

Note: See "Creating Icons for the Desktop" in the *Advanced Users & System Administrator's Guide* for more information about the desktop icon files.

For bitmaps, there is an additional file that is used as a mask, and its extension ends with _m.bm. Thus, there can be a total of three files for each size icon. Here are the icon files for the calculator:

```
Dtcalc.t.bm
Dtcalc.t.pm
Dtcalc.t_m.bm
Dtcalc.m.bm
Dtcalc.m.pm
Dtcalc.m_m.bm
Dtcalc.l.bm
Dtcalc.l.pm
Dtcalc.l_m.bm
```

Note: There are no small icons (Dtcalc.s.bm, Dtcalc.s.pm, Dtcalc.s_m.bm) for the Calculator.

DtActionIcon() returns only a base name; for the Calculator it is Dtcalc. You must choose the type (pixmap or bitmap) and size (tiny, small, medium, or large) and append the applicable extension to the base name. In addition, you must know where the file resides.

To Get the Localized Label for an Action

- Use the DtActionLabel() function to get the localized label for an action.

```
char *DtActionLabel(char *actionName)
```

An action definition may include a label. The label is defined using the label_text field:

```
ACTION action_name
{
   LABEL  label_text
   °
}
```

This label is used in graphical components (such as File Manager and the Application Manager) to label the action's icon. If an action definition does not include a label_text field, the action_name is used.

The value of label_text string should be used by all interface components to identify the action to the end user.

The DtActionLabel() function returns the value of the label_text field in the action definition of the action named actionName. If the label_text field does not exist, the function returns the actionName.

```
labelString = XmStringCreateLocalized(”On File:”);
n = 0;
XtSetArg(args[n], XmNlabelString, labelString); n++;
w = XmCreateLabel(workArea, ”fileLabel”, args, n);
XtManageChild(w);
XmStringFree(labelString);
```

Invoking Actions

After your application has initialized the Desktop Services Library it can then invoke an action.

To Invoke an Action

- Use the DtActionInvoke() function to invoke an action.

```
DtActionInvoke(widget, action, args, argCount, termOpts, execHost,
                                        contexDir, useIndicator,
statusUpdateCb, client_data)
```

DtActionInvoke() searches the action database for an entry that matches the specified action name, and accepts arguments of the class, type and count provided. Remember that your application must initialize and load the database before invoking an action.

DtActionInvoke() Example

The following code segment is from activateCB()(the activate callback for the drawn button) in actions.c.

```
DtActionInvocationID actionId;
/* If a file was specified, build the file argument list */
printf(”%s(%s)\n”,action,file);
if (file != NULL && strlen(file) != 0) {
   ap = (DtActionArg*) XtCalloc(1, sizeof(DtActionArg));
ap[0].argClass = DtACTION_FILE;
ap[0].u.file.name = file;
nap = 1;
   }
    /* Invoke the specified action */
actionId =
DtActionInvoke(shell,action,ap,nap,NULL,NULL,NULL,True,NULL,NULL)
;
}
```

Listing for actions.c

```
/* Include File Declarations */
#include <Xm/XmAll.h>
#include <Dt/Dt.h>
#include <Dt/Action.h>

#define ApplicationClass ”Dtaction”
static Widget shell;
static XtAppContext appContext;
static Widget actionText;
static Widget fileText;
```

```
static void CreateWidgets(Widget);
static void InvokeActionCb(Widget, XtPointer, XtPointer);
static void InvokeAction(char*, char*);
static void DbReloadProc(XtPointer);

void main(int argc, char **argv)
{
    Arg args[20];
    int n=0;
    int numArgs = 0;

    shell = XtAppInitialize(&appContext , ApplicationClass, NULL,
                            0, &argc, argv, NULL, args, n);

    CreateWidgets(shell);

    if (DtInitialize(XtDisplay(shell), shell, argv[0],
        ApplicationClass)==False) {
        /* DtInitialize() has already logged an appropriate error
           msg */
        exit(–1);
    }

    /* Load the filetype/action databases */
    DtDbLoad();

    /* Notice changes to the database without needing to restart
       application */
    DtDbReloadNotify(DbReloadProc, NULL);

    XtRealizeWidget(shell);
    XmProcessTraversal(actionText, XmTRAVERSE_CURRENT);

    XtAppMainLoop(appContext);
}
static void CreateWidgets(Widget shell)
{
    Widget messageBox, workArea, w;
    Arg args[20];
    int n;
    XmString labelString;

    labelString = XmStringCreateLocalized(”Invoke”);

    n = 0;
    XtSetArg(args[n], XmNdialogType, XmDIALOG_TEMPLATE); n++;
    XtSetArg(args[n], XmNokLabelString, labelString); n++;
    messageBox = XmCreateMessageBox(shell, ”messageBox”, args,n);
    XtManageChild(messageBox)
    XmStringFree(labelString);
    XtAddCallback(messageBox, XmNokCallback, InvokeActionCb,
    NULL);

    n = 0;
    XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
    XtSetArg(args[n], XmNpacking, XmPACK_COLUMN); n++;
    XtSetArg(args[n], XmNnumColumns, 2); n++;
    XtSetArg(args[n], XmNentryAlignment, XmALIGNMENT_END); n++;
    workArea = XmCreateWorkArea(messageBox, ”workArea”, args, n);
```

```
    XtManageChild(workArea);

    labelString = XmStringCreateLocalized("Invoke Action:");
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    w = XmCreateLabel(workArea, "actionLabel", args, n);
    XtManageChild(w);
    XmStringFree(labelString);

    labelString = XmStringCreateLocalized("On File:");
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    w = XmCreateLabel(workArea, "fileLabel", args, n);
    XtManageChild(w);
    XmStringFree(labelString);

    n = 0;
    XtSetArg(args[n], XmNcolumns, 12); n++;
    actionText = XmCreateTextField(workArea, "actionText", args,
    n);

    XmProcessTraversal(actionText, XmTRAVERSE_CURRENT);

    XtAppMainLoop(appContext);
}

static void CreateWidgets(Widget shell)
{
    Widget messageBox, workArea, w;
    Arg args[20];
    int n;
    XmString labelString;

    labelString = XmStringCreateLocalized("Invoke");

    n = 0;
    XtSetArg(args[n], XmNdialogType, XmDIALOG_TEMPLATE); n++;
    XtSetArg(args[n], XmNokLabelString, labelString); n++;
    messageBox = XmCreateMessageBox(shell, "messageBox", args,
    n);
    XtManageChild(messageBox);
    XmStringFree(labelString);
    XtAddCallback(messageBox, XmNokCallback, InvokeActionCb,
    NULL);

    n = 0;
    XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
    XtSetArg(args[n], XmNpacking, XmPACK_COLUMN); n++;
    XtSetArg(args[n], XmNnumColumns, 2); n++;
    XtSetArg(args[n], XmNentryAlignment, XmALIGNMENT_END); n++;
    workArea = XmCreateWorkArea(messageBox, "workArea", args, n);
    XtManageChild(workArea);

    labelString = XmStringCreateLocalized("Invoke Action:");
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    w = XmCreateLabel(workArea, "actionLabel", args, n);
    XtManageChild(w);
    XmStringFree(labelString);
```

```
    labelString = XmStringCreateLocalized("On File:");
    n = 0;
    XtSetArg(args[n], XmNlabelString, labelString); n++;
    w = XmCreateLabel(workArea, "fileLabel", args, n);
    XtManageChild(w);
    XmStringFree(labelString);

    n = 0;
    XtSetArg(args[n], XmNcolumns, 12); n++;
    actionText = XmCreateTextField(workArea, "actionText", args,
    n);

    DtActionInvocationID actionId;

    /* If a file was specified, build the file argument list */

    printf("%s(%s)\n",action,file);
    if (file != NULL && strlen(file) != 0) {
        ap = (DtActionArg*) XtCalloc(1, sizeof(DtActionArg));
        ap[0].argClass = DtACTION_FILE;
        ap[0].u.file.name = file;
        nap = 1;
    }

    /* Invoke the specified action */

    actionId = DtActionInvoke(shell, action, ap, nap, NULL, NULL,
    NULL, True, NULL, NULL);
}
```

Accessing the Data-Typing Database

This section describes the data-typing functions and how to use the data- typing database.

- Summary
- Data Criteria and Data Attributes
- Data-Typing Functions
- Registering Objects as Drop Zones
- Example of Using the Data-Typing Database

Summary

Data typing provides an extension to the attributes of files and data beyond what is provided by the traditional UNIX file systems. These extensions consist of attributes, such as icon names, descriptions, and actions, that can be performed on files and data. This information is stored in name/value pairs in the **DATA_ATTRIBUTES** table (or database). The desktop uses a certain set of **DATA_ATTRIBUTES**, described in the following paragraphs. The **DATA_ATTRIBUTES** table is extendable for future and application-specific growth, although extending this table is not recommended because other applications may not check the additions.

Data is matched with a specific file or data entry in a **DATA_CRITERIA** table. The **DATA_CRITERIA** table entries are sorted in decreasing order from most specific to least specific. For example, **/usr/lib/lib*** is more specific than **/usr/*** and would, therefore, appear first. When a request to type a file or data is made, the table is checked in sequence to find the best match using the information provided either from the file or from the data. When an information and entry match is found, **DATA_ATTRIBUTES_NAME** is used to find the proper **DATA_ATTRIBUTES** entry.

If you want your application to present data objects (either files or data buffers) to the user in a manner consistent with the desktop, use the **DtDts*** API to determine how to display the data objects and how to operate on them. For example, your application can determine the icon that represents a data object by calling the **DtDtsDataTypeToAttributeValue** function for the ICON attribute.

Library and Header Files

To use data typing, you need to link to the **libDtSvc** library. Actions are usually loaded with the data-typing information. Actions require links to the **libXm** and **libX11** libraries. The header files are **Dt/Dts.h** and **Dt/Dt.h**.

Demo Program

A demo program containing an example of how to use the data-typing database is in **/usr/dt/examples/dtdts/datatypes/datatyping.c**.

Using Data Typing

To Type Your Application Using the Database

- Create an *<app>***.dt** database file using Create Actions (**dtcreate**).

See the *Common Desktop Environment: Advanced User's and System Administrator's Guide* for information on using Create Actions.

Data Criteria and Data Attributes

Data typing consists of two parts:

- A database that stores data criteria and data attributes
- A collection of routines that query the database

The attributes of data criteria, in alphabetical order, are:

- **CONTENT**
- **DATA_ATTRIBUTES_NAME**
- **LINK_NAME**
- **LINK_PATH**
- **MODE**
- **NAME_PATTERN**
- **PATH_PATTERN**

The following table describes the data criteria in the order in which you are most likely to use them.

Data Criteria in Order of Most Likely Use

Criteria	Description	Typical Usage
`DATA_ATTRIBUTES_NAME`	The name of this type of data. This value is a record_name in the data attributes table.	`POSTSCRIPT`
`NAME_PATTERN`	A shell pattern-matching expression describing the file names that could match this data. The default is an empty string, which means to ignore file patterns in matching.	`*.ps`
`CONTENT`	Three values that are interpreted as the start, type, and value fields of the magic file used by the file utility. See the file(1) man page for more information. The default is an empty field, which means to ignore contents in matching. The following types are examples of what can be matched: string, byte, short, long, and file name.	0 string !%
`MODE`	A string of zero to four characters that match the mode field of a stat structure. See the stat(2) man page for more information. The first character indicates: d matches a directory s matches a socket l matches a symbolic link f matches a regular file b matches a block file c matches a character special file	`f&!x`

Data Criteria in Order of Most Likely Use

Criteria	Description	Typical Usage
	The characters listed below can be either the first or subsequent characters: r matches any file with any of its user, group, or other read permission bits set. w matches any file with any of its user, group, or other write permission bits set. x matches any file with any of its user, group, or other execute or directory-search permission bits set. For example, the MODE field of frw matches any regular file that is readable or writable; x matches any file with any of its executable or search bits set. The default is an empty field, which means to ignore the mode in matching.	
PATH_PATTERN	A shell pattern-matching expression describing the absolute path names that could match this data. The default is an empty string, which means to ignore path patterns in matching.	*/mysubdir/*
LINK_NAME	See dtdtsfile(4) man page.	
LINK_PATH	See dtdtsfile(4) man page.	

Some of the more common attributes of data types, in alphabetical order, are:

- **ACTIONS**
- **COPY_TO_ACTION**
- **DESCRIPTION**
- **ICON**
- **INSTANCE_ICON**
- **IS_EXECUTABLE**
- **IS_TEXT**
- **LINK_TO_ACTION**
- **MEDIA**
- **MIME_TYPE**
- **MOVE_TO_ACTION**
- **NAME_TEMPLATE**
- **PROPERTIES**
- **X400_TYPE**

The following table describes the data attributes in the order in which you are most likely to use them.

Data Attributes in Order of Most Likely Use

Criteria	Description	Typical Usage
`DESCRIPTION`	A human-readable description of this data. If this field is NULL or is not included in the data attribute record, the name of the data attribute should be used.	This is a PostScript page description.
`ICON`	The name of the icon to be used for this data. If this field is NULL or is not included in the data attribute record, the standard icon should be used. See dtdtsfile(4) for more details on icon naming.	`Dtps`
`PROPERTIES`	Keywords to indicate properties for this data. Valid values are invisible and visible. If this field is NULL or is not included in the data attribute record, the visible property should be assumed. Use this if you want to completely hide files from the user.	`invisible`
`ACTIONS`	A list of actions that can be performed on this data. This list refers to names in the action table for actions that are to be presented to the user for objects of this type. If this field is NULL or is not included in the data attribute record, no action is available.	Open,Print

Data Attributes in Order of Most Likely Use

Criteria	Description	Typical Usage
NAME_TEMPLATE Field	A string used to create a new file for data of this type. The string is passed to sprintf(3) with the file name as the single argument. The default is empty. Contrast this field with the NAME_PATTERN field of the data criteria table in that the template is used to create a specific file, such as %s.c, whereas the pattern is used to find files, such as *.c.	%s.ps
IS_EXECUTABLE Field	A string-Boolean value that tells users of this data type that it can be executed as an application. If IS_EXECUTABLE is set to true (see DtDtsIsTrue()) the data is executable. If this field is NULL, is not included in the data attribute record, or is not set to true, then the data is considered not executable.	true
MOVE_TO_ACTION	The name of an action to be invoked when an object is moved to the current object.	FILESYSTEM_MOVE
COPY_TO_ACTION	The name of an action to be invoked when an object is copied to the current object.	FILESYSTEM_COPY
LINK_TO_ACTION	The name of an action to be invoked when an object is linked to the current object.	FILESYSTEM_LINK
IS_TEXT	A string-Boolean value that tells users of this data type that it is suitable for manipulation (viewing or editing) in a text editor or text widget. The IS_TEXT field is set to true (see DtDtsIsTrue()) if the data is textual in nature and if it should be presented to the user in text form. Criteria for making this determination include whether data consists of human language, is generated and maintained manually, is usefully viewable and editable in a text editor, or contains no (or only minimal) structuring and formatting information. If the IS_TEXT field is true, the data is eligible to be displayed directly by an application. That is, the application can load the data directly into a text editing widget, such as XmText.	See IS_TEXT Attribute Examples table.
MEDIA Field	The names in the MEDIA name space describe the form of the data itself. MEDIA names are used as ICCCM selection targets, named in the MEDIA field of the data-type records, and used in the type parameter of ToolTalk Media Exchange messages. The MEDIA name space is a subset of the name space of selection target atoms as defined by the ICCCM. All selection targets that specify a data format are valid MEDIA names, and all valid MEDIA names can be used directly as selection targets. Some selection targets specify an attribute of the selection (for example, LIST_LENGTH) or a side effect to occur (for example, DELETE), rather than a data format. These attribute selection targets are not part of the MEDIA name space.	POSTSCRIPT

Data Attributes in Order of Most Likely Use

Criteria	Description	Typical Usage
MIME_TYPE	MEDIA is the desktop internal, unique name for data types. However, other external naming authorities have also established name spaces. Multipurpose Internet Message Extensions (MIME), as described in the referenced MIME RFC, is one of those external registries, and is the standard-type name space for the desktop mailer.	application/postscript
X400_TYPE	X.400 types are similar in structure to the MEDIA type, but are formatted using different rules and have different naming authorities.	1 2 840 113556 3 2 850
INSTANCE_ICON Field	The name of the icon to be used for this instance of data, typically a value such as %name%.icon [Bug in dtdtsfile(4) man page, too.] If INSTANCE_ICON is set, the application should use it instead of ICON. If this field is NULL or is not included in the data attribute record, the ICON field should be used.	/myicondir/%name%.bm
DATA_HOST	The DATA_HOST attribute is not a field that can be added to the data attributes table in the *.dt file, but it may be returned to an application reading attributes from the table. The data-typing service adds this attribute automatically to indicate the host system from which the data type was loaded. If this field is NULL or is not included in the data attribute record, the data type was loaded from the local system.	

The IS_TEXT field differs from the text attribute of the MIME_TYPE field, which is the MIME content type, as described in the referenced MIME_ RFC. The MIME content type determines whether the data consists of textual characters or byte values. If the data consists of textual characters, and the data is labeled as text/*, the IS_TEXT field determines whether it is appropriate for the data to be presented to users in textual form.

The following table shows some examples of IS_TEXT usage with different MIME_TYPE attributes.

IS_TEXT Attribute Examples

Description and MIME_TYPE Attribute	IS_TEXT Value
Human language encoded in ASCII with MIME_TYPE text/plain	IS_TEXT true
Human language encoded in E*UC, JIS, Unicode, or an ISO Latin charset with MIME_TYPE text/plain; charset=XXX	IS_TEXT true
CalendarAppointmentAttrs with a MIME_TYPE text/plain	IS_TEXT false
HyperText Markup Language (HTML) with a MIME_TYPE text/html	IS_TEXT true
PostScript with MIME_TYPE application/postscript	IS_TEXT false
C program source (C_SRC) with MIME_TYPE text/plain	IS_TEXT true

IS_TEXT Attribute Examples	
Description and MIME_TYPE Attribute	IS_TEXT Value
Bitmaps and pixmaps (XBM and XPM) with MIME_TYPE text/plain	IS_TEXT false
Project or module files for the desktop application building service with MIME_TYPE text/plain	IS_TEXT false
Shell scripts with MIME_TYPE text/plain	IS_TEXT false
Encoded text produced by uuencode(1) with MIME_TYPE text/plain	IS_TEXT false
*MIME_TYPE text/plain	IS_TEXT false

See the **dtdtsfile(4)** man page for more information about data-type attributes.

Data-Typing Functions

To look up an attribute for a data object, you must first determine the type of the object and then ask for the appropriate attribute value for that type. The functions that you can use to query the database for data information are shown in the following table. Each of these functions has a man page in section (3). Refer to the appropriate man page for more information.

Data-Typing Database Query Functions	
Function	Description
`DtDtsBufferToAttributeList`	Finds the list of data attributes for a given buffer.
`DtDtsBufferToAttributeValue`	Finds the data attribute for a given buffer.
`DtDtsBufferToDataType`	Finds the data-type name for a given buffer.
`DtDtsDataToDataType`	Finds the data type for a given set of data.
`DtDtsDataTypeIsAction`	Returns the resulting saved data type for the directory.
`DtDtsDataTypeNames`	Finds a complete list of available data types.
`DtDtsDataTypeToAttributeList`	Finds the attribute list for a given data attribute name.
`DtDtsDataTypeToAttributeValue`	Finds the attribute value for a given data attribute name.
`DtDtsFileToAttributeList`	Finds the list of data attributes for a given file.
`DtDtsFileToAttributeValue`	Finds the data attribute value for a given file.
`DtDtsFileToDataType`	Finds the data type for a given file.

Data-Typing Database Query Functions	
Function	Description
`DtDtsFindAttribute`	Finds the list of data types where attribute name matches value.
`DtDtsFreeAttributeList`	Frees the memory of the given attribute list.
`DtDtsFreeAttributeValue`	Frees the memory of the given attribute value.
`DtDtsFreeDataType`	Frees the application memory for the given data-type name.
`DtDtsFreeDataTypeNames`	Releases memory created with the DtDtsDataTypeNames or DtDtsFindAttribute call.
`DtDtsIsTrue`	A convenience function that converts a string to a Boolean.
`DtDtsRelease`	Unloads the data-typing database information, generally in preparation for a reload.
`DtDtsSetDataType`	Sets the data type for the specified directory.
`DtsLoadDataTypes`	Initializes and loads the database fields for the data-typing functions. Use instead of DtDbLoad when you do not need to use actions or action types and you need extra performance. Use DtDbLoad when you need to use actions.

You can type data and retrieve attributes in one of three ways: simple, intermediate, or advanced.

Simple Data Typing

The simplest way to type data is to use the following functions:

- **DtDtsFileToAttributeList**
- `DtDtsFileToAttributeValue`

When you use these functions, a file is typed and a single attribute, or the entire list, is retrieved. System calls are made, data is typed, and the attribute is retrieved. These functions call the intermediate data-typing functions.

- `DtDtsBufferToAttributeList`
- `DtDtsBufferToAttributeValue`

Buffers are assumed to have a mode that matches regular files that have read/write permissions. See “Advanced Data Typing” to type read-only buffers.

Intermediate Data Typing

When you type data and retrieve attributes, the data-typing part of the process is the most expensive in terms of performance. You can type data in a second way that improves performance by separating the data-typing and attribute-retrieval functions. Use the following functions for intermediate data typing:

- `DtDtsBufferToDataType`
- `DtDtsFileToDataType`
- `DtDtsDataTypeToAttributeList`
- DtDtsDataTypeToAttributeValue

Use these functions if your application queries for more than a single attribute value. When you use these functions, an object is typed and then that type is used to retrieve one or more attributes from the attribute list.

Using the intermediate data-typing functions is the recommended way to type data and retrieve attributes. These functions call the advanced data-typing functions and make the same assumptions about buffers as the simpler data typing.

Advanced Data Typing

Advanced data typing separates system calls, data typing, and attribute retrieval even further. Advanced data typing is more complicated to code because it uses data from existing system calls, which are initialized in advance and are not included as part of the data-typing function. Use the following function for advanced data typing:

DtDtsDataToDataType

To type a read-only buffer, a stat structure should be passed that has the st_mode field set to S_IFREG | S_IROTH | S_IRGRP | S_IRUSR.

Data Types That Are Actions (DtDtsDataTypeIsAction)

For every action in a database a *synthetic data type* is generated when a database is loaded that allows actions to be typed. These data types may have two additional attributes:

- **IS_ACTION** is a string-Boolean value that tells users of this data type that it is an action. If **IS_ACTION** is set to the string **true** (independent of case), the data is an action.
- **IS_SYNTHETIC** is a string-Boolean value that tells users of this data type that it was generated from an entry in the **ACTION** table. If **IS_SYNTHETIC** is set to **true**, the data type was generated.

Registering Objects as Drop Zones

If your application defines icons for data objects, you may choose to support those icons as drop zones. If so, you need to query the MOVE_TO_ACTION, COPY_TO_ACTION, and LINK_TO_ACTION attributes to determine the drop behavior for those objects. Objects should support drop operations (for example, copy, move, and link) only if the corresponding attribute value is not NULL. If all three attributes have NULL values, the object should not be registered as a drop site.

When you write your desktop application, follow these steps to ensure that it provides all the drag and drop behavior that you intend:

1.. In your application, decide if you need to define any data types.

2.. For each data type you define, decide whether you want the associated object to be a drop zone.

3.. For each object that you want to register as a drop zone, decide which operations—move, copy, or link—you want to define.

4.. For the drop operations that are valid for each object, define the appropriate drop actions (set the MOVE_TO_ACTION, COPY_TO_ACTION, and LINK_TO_ACTION attributes).

If you want an object of a certain type in your application to be a drop zone, you need to decide which drop operations to define for the object: move, copy, link, or some combination of the three.

Every desktop data object should have three associated drop attributes:

- MOVE_TO_ACTION
- COPY_TO_ACTION
- LINK_TO_ACTION

If you want to define a move operation for the object, set the MOVE_TO_ACTION attribute to reflect the appropriate move behavior. If you want to define a copy operation, set the COPY_TO_ACTION attribute. If you want to define a link operation, set the LINK_TO_ACTION attribute.

Whenever you set at least one of these attributes for an object with a defined data type, your application registers that object as a drop zone. The application sets the Motif resource XmNdropSiteOperations to the appropriate operation (XmDROP_MOVE, XmDROP_COPY, or XmDROP_LINK) corresponding to the drop attribute you have defined.

Using these three attribues, you can change defaults or define new move, copy, and link behavior for your data type.

Note: If, for an object with an associated data type, you do not define values for any of these three drop attributes, that object is not registered as a drop zone.

When a user drags an object to a drop zone, your application determines which gesture (that is, which drag operation) was used to make the drop. Based on the drag operation and the drop zone's data type, the application retrieves a drop attribute from the data–types database. It then calls DtActionInvoke, using the following two rules to determine its parameters:

1.. If the user drops objects A and B onto object C, call DtActionInvoke with these parameters:

```
DtActionInvode(drop_action_name, C, A, B)
```

2.. The previous rule has one exception: If object C is an action, the parameter list does not include C:

```
DtActionInvode(drop_action_name, A, B)
```

where drop_action_name is one of MOVE_TO_ACTION, COPY_TO_ACTION, or LINK_TO_ACTION.

The File Manager, along with its directory and folder objects, exemplifies how the desktop uses the move, copy, and link drop attributes. A user can drag and drop objects (files) to directory folders. File Manager defines MOVE_TO_ACTION, COPY_TO_ACTION, and LINK_TO_ACTION actions for folder objects. These actions perform the appropriate file system move, copy, and link system functions. If you want, you can redefine the attributes to perform different move, copy, and link functions.

See /usr/dt/appconfig/types/C/dtfile.dt for an example of how to define the MOVE_TO_ACTION, COPY_TO_ACTION, and LINK_TO_ACTION attributes. See "Integrating with Drag and Drop," for information about how to use drag and drop.

Example of Drop Types

Motif determines the type of a drop (move, copy, or link) using the formula shown in the following table:

Types of Drags	
User Action	Result
`Drop + Shift`	Move drop
`Drop + Ctrl`	Copy drop
`Drop + Shift + Ctrl`	Link drop

The following drop defaults to the first type that a drop site is registered for using the following precedence:

1.. Move

2.. Copy

3.. Link

For example, if a drop site is registered for Move and Link drops and a user dropped an object on this site without using a modifier key, Motif defaults to a Move drop. If a drop site is registered for Copy and Link drops, Motif defaults to a Copy drop.

Note: Actual Motif terminology for modifiers keys: is
<Shift>Btransfer = MOVE
<Ctrl>BTransfer = COPY
<Shift><Ctrl>BTransfer = Link
You might want to use this terminology instead.

Example of Using File Manager Move, Copy, Link Feature

When File Manager creates a view of a file, it queries the Types database for three attributes for each file:

- MOVE_TO_ACTION
- COPY_TO_ACTION
- LINK_TO_ACTION

These attributes tell the File Manager what action to take for a move drop, a copy drop, and a link drop respectively. All objects that have values for at least one of these attributes are registered as drop sites. The objects are registered to receive only those drops that have a corresponding action in the Types database.

In the following example, all executable objects are registered as drop sites that receive copy drops only. When a user drops an object or objects on an executable using:

- No modifier keys, or
- The Control modifier key

the drop is accepted. A File Manager callback is triggered. The File Manager callback starts the Execute action. All dropped files are sent as input to the Execute action.

```
DATA_ATTRIBUTES EXECUTABLE
{
        ACTIONS          Run, Open
        ICON             Dtexec
        IS_EXECUTABLE    true
        COPY_TO_ACTION   Execute
        MIME_TYPE        application/octet-stream
        DESCRIPTION      This file contains a shell script or a \
                         compiled program that can be executed. \
                         Its data type is named EXECUTABLE.
}
```

In the following example, all editor objects are registered as drop sizes that receive move and copy drops. When a user drops an object or objects on an editor object using:

- No modifier keys, or
- The Shift modifier key, or
- The Control modifier key

the drop is accepted. A File Manager callback is triggered. For an unmodified drop or a Shift modified drop, File Manager starts the Move action, Edit_Write. This action enables the user to edit the dropped files and save the results. For a Control modified drop, File Manager starts the Copy action, Edit_ReadOnly. This action enables the user to edit the dropped files but does no enable the user to save the results.

```
DATA_ATTRIBUTES EDITOR
{
        .
        .
        .
        MOVE_TO_ACTION   Edit_Write
        COPY_TO_ACTION   Edit_ReadOnly
}
```

Example of Using the Data-Typing Database

This section contains example code of how to use data typing. You can find this example code in /usr/dt/examples/dtdts/datatyping.c. The example code displays the data type, icon name, and supported actions for each file passed to it. You can then use the dtaction client to run a supported action on the file. The usage for datatyping is:

```
datatyping file1 [file2 ...]
#include <Xm/Form.h>
#include <Xm/Text.h>
#include <Dt/Dts.h>

#define ApplicationClass ”DtDatatyping”

static Widget text;

static void DisplayTypeInfo(int, char**);

int main(int argc, char **argv)
{
    XtAppContext appContext;
```

```
    Widget toplevel, form;
    Arg args[20];
    int n;

    toplevel = XtAppInitialize(&appContext, ApplicationClass,
    NULL, 0, argc, argv, NULL, NULL, 0);

    if (argc == 1) {
        printf("%s: No files specified.\n", argv[0]);
        exit(1);
    }

    form = XmCreateForm(toplevel, "form", NULL, 0);
    XtManageChild(form);
    n = 0;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNeditable, False); n++;
    XtSetArg(args[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg(args[n], XmNrows, 25); n++;
    XtSetArg(args[n], XmNcolumns, 90); n++;
    text = XmCreateScrolledText(form, "text", args, n);
    XtManageChild(text);

    XtRealizeWidget(toplevel);
      if (DtAppInitialize(appContext, XtDisplay(toplevel),
      toplevel, rgv[0], ApplicationClass) == False) {
        printf("%s: Couldn't initialize Dt\n", argv[0]);
        exit(1);
    }

    DtDbLoad();

    DisplayTypeInfo(argc, argv);

    XtAppMainLoop(appContext);
}
```

```
static void DisplayTypeInfo(int argc, char **argv)
{
    char *file;
    char *datatype;
    char *icon;
    char *actions;
    char str[100];
    int i;

    sprintf(str, "%-30s\t%-10s\t%-8s\t%-20s\n",
                "File",
                "DataType",
                "Icon",
                "Actions");
    XmTextInsert(text, XmTextGetLastPosition(text), str);

    sprintf(str, "%-30s\t%-10s\t%-8s\t%-20s\n",
                "------------------",
                "--------",
                "----",
                "-------");
    XmTextInsert(text, XmTextGetLastPosition(text), str);

        for(i=1; i < argc; i++) {
        char *file = argv[i];

        /* find out the Dts data type */
        datatype = DtDtsFileToDataType(file);

        if(datatype) {
            /* find the icon attribute for the data type */
            icon = DtDtsDataTypeToAttributeValue(datatype,
            DtDTS_DA_ICON, file);
        }

        /*  Directly find the action attribute for a file */

        actions = DtDtsFileToAttributeValue(file,
                        DtDTS_DA_ACTION_LIST);
```

```
    sprintf(str, "%-30s\t%-10s\t%-8s\t%s\n",
                    file,
                    datatype?datatype:"unknown",
                    icon?icon:"unknown",
                    actions?actions:"unknown");
    XmTextInsert(text, XmTextGetLastPosition(text), str);

    /* Free the space allocated by Dts */

    DtDtsFreeAttributeValue(icon);
    DtDtsFreeAttributeValue(actions);
    DtDtsFreeDataType(datatype);
}
```

Integrating with Calendar

The Calendar application program interface (API) provides a programmatic way to access and manage calendar data in a networked environment. The API supports inserting, deleting, and modifying of entries as well as browse and find capabilities. It also supports calendar administration functions.

The Calendar API is an implementation of the X.400 Application Programming Interface Association's (XAPIA) Calendaring and Scheduling API (CSA API). CSA API defines a set of high-level functions so that applications that are calendar enabled can access the varied features of the calendaring and scheduling service. For more information about the latest XAPIA Specification, contact the X.400 API Association, 800 El Camino Real, Mountain View, California 94043.

This section describes the Calendar API in these sections:

- Library and Header Files
- Demo Program
- Using the Calendar API
- Overview of the CSA API
- Functional Architecture
- Data Structures
- Calendar Attributes
- Entry Attributes
- General Information about Functions
- Administration Functions
- Calendar Management Functions
- Entry Management Functions

Library and Header Files

To use the Calendar API, you need to link to the **libcsa** library. The header file is **csa/csa.h**.

Demo Program

A demo program containing an example of how to use the Calendar API is in **/usr/dt/examples/dtcalendar.**

Using the Calendar API

How to Integrate with Calendar

The Calendar API provides a way to access and manage calendar data in a networked environment.

1.. Include **csa/csa.h** in your application.

2.. Use the calendar API to incorporate the calendar operations you want in your application.

3.. Link with **libcsa.**

Overview of the CSA API

The CSA interface enables a common interface to a calendaring and scheduling service. For each CSA implementation, the view and capabilities presented by CSA must be mapped to the view and capabilities of the underlying calendaring service. The interface is designed to be independent of the actual calendaring and scheduling implementation. The interface is also designed to be independent of the operating system and underlying hardware used by the calendaring service.

The number of function calls provided is minimal. A single set of functions manage multiple types of calendar entries.

C Naming Conventions

The identifier for an element of the C interface is derived from the generic name of the element and its associated data type, as specified in the following table. The generic name is prefixed with the character string in the second column of the table; alphabetic characters are converted to the case in the third column.

Derivation of C Naming Conventions

Element Type	Prefix	Case
Data type	**CSA_**	Lower
Data value	**CSA_**	Upper
Function	**csa_**	Lower
Function argument	**none**	Lower
Function result	**none**	Lower
Constant	**CSA_**	Upper
Error	**CSA_E_**	Upper
Macro	**CSA_**	Upper
Reserved for extension sets	**CSA_XS_**	Any
Reserved for extensions	**CSA_X_**	Any
Reserved for use by implementors	**CSAP**	Any
Reserved for vendor function extensions	**csa_x**	Lower
Structure Tag	**CSA_TAG_**	Upper

Elements with the prefix **CSAP** (any case) are reserved for internal proprietary use by implementors of the CSA service. They are not intended for direct use by programs written using the CSA interface.

The prefixes **CSA_XS_**, **CSA_X_** (in either uppercase or lowercase), and **csa_x** are reserved for extensions of the interface by vendors or groups. The specification defines these interface extensions as extensions to the base set of functions.

For constant data values, an additional string is usually appended to **CSA_** to indicate the data structure or function for the constant data value.

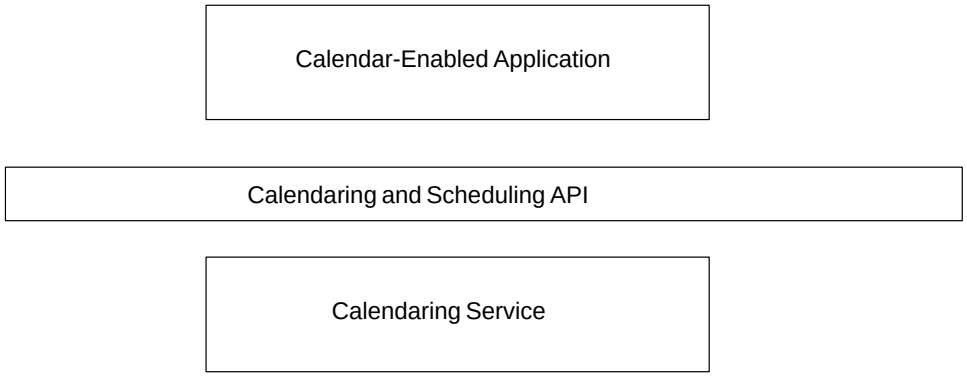
Functional Architecture

This section describes the functional architecture of services supporting the CSA API. It provides an abstract implementation model, an abstract data model, and a functional overview.

Implementation Model

The abstract implementation model is provided as a reference aid to help you understand the scope of the CSA API.

The CSA interface is defined between a calendar-enabled application and a calendaring service. All functions in this interface are designed to be independent of the calendaring service; however, the API does allow protocol-specific extensions to the common functions to be invoked through the use of extensions. See “Extensions” for more information. The relationship of the CSA interface to a calendar-enabled application and the calendar service is shown the following figure.



Positioning of the Calendaring and Scheduling API

The model of the CSA interface can be divided into three components: administration, calendar management, and entry management. These components are shown in the following figure.

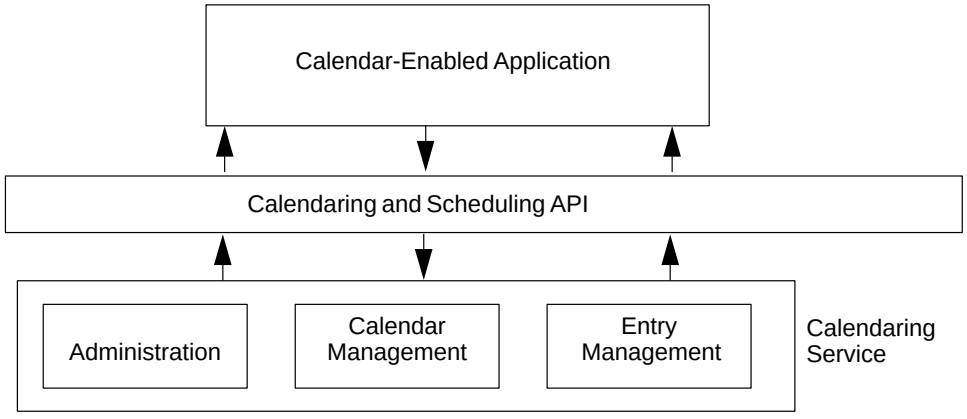


Figure 1. Components of the Calendaring and Scheduling API

Access to the calendaring service is established through a calendaring session. The session provides for a valid connection to the calendaring service and assists in ensuring the integrity of the calendaring information maintained by the service. A calendar-enabled application logs on to an individual calendar within the calendaring service to establish a

valid session or connection. The session is terminated by the calendar-enabled application logging off from the calendar.

The calendaring service maintains one or more calendars. The calendar service provides some level of administration support for these calendars. A calendar-enabled application might access a list of the calendars maintained by a particular calendar service. In addition, the calendar service might provide support for the archive and restore of calendar information into some implementation-specific, persistent format. Where a calendar service provides support for maintaining more than one calendar, support functions are defined for creating and deleting calendars. In addition, functions are provided to support administering the characteristics of the calendar.

The majority of the functions within the CSA interface manage individual calendar entries. Calendar entries may be either events, to dos, or memos. Entries can be added, deleted, updated, and read from a particular calendar. A calendar-enabled application can also add reminders to calendar entries.

Data Model

The CSA interface is an access method to a conceptual back-end store of calendaring information that is maintained by a calendaring service. A common data model is helpful in visualizing the components of the calendaring information maintained by a calendaring service.

Calendar Entities

The data model is based on the concept of a calendar entity. The calendar is represented by a named collection of administrative calendar attributes and calendar entries. A calendar is owned by an individual user. The user could represent a person, a group of people, or a resource.

The calendar attributes are a set of named values that may represent common, implementation-specific, or application-specific administrative characteristics about the calendar. For example, time zone, name, owner, and access rights to the calendar can be specified in individual calendar attributes.

The calendar entries are the primary components of a calendar. The three classes of calendar entries follow:

- Events
- To Dos
- Memos

Calendar entries are represented by a uniquely named collection of entry attributes. The entry attributes are a set of named values that represent common, implementation-specific, or application-specific characteristics of the calendar entry. For example, an event might contain a start and end date and time, a description, and a subtype. A to do might contain the date it was created, the due date, a priority, and a status. A memo might contain the date it was created and a text content or description.

The calendar attributes and entry attributes consist of a name, type, and value tuple. Common attributes defined by the specification can be extended. Implementations can define implementation-specific attributes. In addition, some implementations may provide the capability for applications to define application-specific attributes. The CDE implementation supports application defined attributes.

Access Rights

The accessibility of a calendar to an individual user can be controlled by the access rights given that user. Access rights are paired with a calendar user. CSA allows users to be an

individual, group, or resource. The CDE implementation only supports individual users. The access rights are maintained in an access list. The access list is a particular calendar attribute. The access rights are individually controlled and can be accumulated to define a range of accessibility of a user to a calendar and its entries. The access rights can also be specified in terms of the following access roles:

- The owner of a calendar
- The organizer of a particular entry within the calendar
- The sponsor of a particular entry within the calendar

The owner role enables the user to do anything to the calendar or calendar entries that the owner of the calendar can do, including deleting the calendar; viewing, inserting, and changing calendar attributes; adding and deleting calendar entries; and viewing, inserting, and changing entry attributes.

The organizer role enables the user to delete the entry or view and change entry attributes of those calendar entries for which the user is specified as the organizer. The organizer defaults to the calendar user who created the entry.

The sponsor role enables the user to delete the entry or view and change entry attributes for those calendar entries for which the user is specified as the sponsor. The sponsor is the calendar user who effectively owns the calendar entry.

In addition to these roles, an access right can be set to limit access to free time searches; view, insert, or change calendar attributes; or view, insert or change entries, depending on whether they are classified as public, confidential, or private. The entry classification acts as a secondary filter on accessibility.

Functional Overview

The CSA interface supports three principle types of tasks:

- Administration
- Calendar management
- Entry management

Administration

Most of the CSA function calls occur within a calendar session. The calendar session is a logical connection between the calendar-enabled application and a particular calendar maintained by the calendaring service. A session is established with a call to the **csa_logon()** function and terminated with a call to the **csa_logoff()** function. The context of the session is represented by a session handle. This handle provides a token in each of the CSA functions to distinguish one calendar session from another. The **csa_logon()** function also authenticates the user to the calendaring service and sets session attributes. Currently, there is no support for sharing calendar sessions among applications.

The **csa_list_calendars()** function is used to list the names of the calendars managed by a particular calendar service.

The **csa_query_configuration()** function is used to list information about the current calendar service configuration. This information can include the character set, line terminator characters for text strings, default service name, default authorization user identifier for the specified calendar service, an indicator of whether a password is needed to authenticate the user identifier, an indicator of whether the common extensions for user interface dialogs is supported, and the CSA specification supported by the implementation.

The CSA implementation provides support for managing the memory for calendar objects and attributes that are returned by the service. The **csa_free()** function is used to free up this

memory after it is no longer needed. It is the responsibility of the application to free up the memory allocated and managed by the calendar service.

Calendar Management

The CSA interface provides several calendar management functions. The CDE implementation supports multiple calendars per calendar service; the calendar-enabled application can add or delete calendars. The **csa_delete_calendar()** function is used to delete calendars. The **csa_add_calendar()** function is used to add new calendars to the service.

The application can also list, read, and update calendar attributes using the **csa_list_calendar_attributes()**, **csa_read_calendar_attributes()**, and **csa_update_calendar_attributes()** functions. The application can register callback functions for receiving notification of a calendar logon, calendar deletion, update of calendar attributes, addition of a new calendar entry, deletion of a calendar entry, and update of a calendar entry. The callback function is only registered for the duration of the calendar session. In any case, this information may be invaluable for some calendar administration applications.

Entry Management

The CSA interface has a robust set of functions for managing calendar entries. The context of a calendar entry in a calendar session is maintained by the entry handle. This handle provides a token in the CSA functions to distinguish one calendar entry from another. The entry handle is returned by the **csa_add_entry()** and **csa_list_entries()** functions. The entry handle is valid for the duration of the calendar session or until the entry is deleted or updated. The entry handle becomes invalid when it is freed by a call to **csa_free()**.

The **csa_add_entry()** function is used to add new entries to a calendar. The **csa_delete_entry()** function is used to delete an entry in a calendar. The **csa_list_entries()** function is used to enumerate the calendar entries that match a particular set of entry attribute criteria. The **csa_read_entry_attributes()** function is used to get either all or a set of entry attribute values associated with a particular calendar entry.

To add an entry to a calendar, a calendar-enabled application must first establish a session with the calendaring service using the **csa_logon()** function. Then the application invokes the **csa_add_entry()** function to specify the new entry. The calendar-enabled application is responsible for composing the attributes used in the **csa_add_entry()** function. The session is terminated using the **csa_logoff()** function.

The entry attributes in an individual calendar entry can be enumerated with the **csa_list_entry_attributes()** function. The values of one or more attributes can be read with the **csa_ read_entry_attributes()** function. Individual entry attributes can be modified with the **csa_update_entry_attributes()** function.

Memory allocated by the CSA implementation for retrieved calendar information is released by passing the associated memory pointers to the **csa_free()** function.

Some calendar entries are associated with a recurring activity. The **csa_list_entry_sequence()** function can be used to enumerate the other recurring calendar entries. This function returns a list of entry handles for the recurring entries.

The CDE calendar server provides support for alarms or reminders to be associated with calendar entries. Reminders can take the form of audio reminders from the terminal speaker, flashing reminders presented on the terminal screen, mail reminders sent to the calendar user, or pop-up reminders presented on the terminal screen. The calendar service manages the reminders, but it is the responsibility of the calendar application to retrieve the reminder information and act on it. The **csa_read_next_reminder()** function is used to read the information about the next scheduled type of reminder.

Extensions

The major data structures and functions defined in the CSA specification can be extended methodically through extensions. Extensions are used to add additional fields to data structures and additional parameters to a function call. A standard generic data structure has been defined for these extensions. It consists of an item code, identifying the extension; an item data, holding the length of extension data or the data itself; an item reference, pointing to where the extension value is stored or **NULL** if there is no related item storage; and flags for the extension.

Extensions that are additional parameters to a function call may be input or output. That is, the extension may be passed as input parameters from the application to the CSA service or passed as output parameters from CSA service to the application. If an extension is an input parameter, the application allocates memory for the extension structure and any other structures associated with the extension. If an extension is an output parameter, the CSA service allocates the storage for the extension result, if necessary. In this case, the application must free the allocated storage with a **csa_free()** call.

If an extension that is not supported is requested, **CSA_E_UNSUPPORTED_FUNCTION_EXT** is returned.

Data Structures

The following table lists the CSA data structures. See the relevant man page for complete information.

CSA Data Structures

Data Type Name	Description
Access List	List of access rights structures for calendar users
Attendee List	List of attendee structures
Attribute	Attribute structure
Attribute Reference	Attribute reference structure
Boolean	A value that indicates logical true or false
Buffer	Pointer to a data item
Calendar User	Calendar user structure
Callback Data Structures	Callback data structures
Date and Time	Date and time designation
Date and Time List	List of date and time values
Date and Time Range	Range of date and time
Entry Handle	Handle for the calendar entry
Enumerated	Data type containing a value from an enumeration
Extension	Extension structure
Flags	Container for bit masks
Free Time	Free time structure

CSA Data Structures	
Data Type Name	Description
Opaque Data	Opaque data structure
Reminder	Reminder structure
Reminder Reference	Reminder reference structure
Return Code	Return value indicating either that a function succeeded or why it failed
Service Reference	Service reference structure
Session Handle	Handle for the calendar session
String	Character string pointer
Time Duration	Time duration

Calendar Attributes

The following table lists the calendar attributes supported in the CDE implementation. See the relevant man page for more information. The list of calendar attributes is extensible through the extended naming convention.

CSA Calendar Attributes		
Attribute Name	Description	Read-Only
Access List	Access list attribute	No
Calendar Name	Calendar name attribute	Yes
Calendar Owner	Calendar owner attribute	Yes
Calendar Size	Calendar size attribute (computed)	Yes
Character Set	Character set attribute	No
Date Created	Calendar creation date attribute	Yes
Number Entries	Number of entries attribute (computed)	Yes
Product Identifier	Product identifier attribute	Yes
Time Zone	Time zone attribute	No
Version	Product version attribute	Yes

The following section provides additional information about the calendar attributes listed in the CSA Calendar Attributes table.

- **Access List**

When a new calendar is added and if no access list is specified, the default access list contains a special user world and its corresponding access right is **CSA_VIEW_PUBLIC_ENTRIES**, which provides access right to list and read calendar entries with public classification. The special user world includes all users.

- **Calendar Name**

The calendar name is specified when the calendar is created using **csa_add_calendar().** It becomes read-only and cannot be changed after the calendar is created.

- **Calendar Owner**

The calendar owner is set to the user who is running the application that calls **csa_add_calendar()** to create the calendar. It becomes read-only and cannot be changed after the calendar is created.

- **Character Set**

The CDE common locale name should be used to set this value.

CDE Calendar Attributes

The followings are CDE-defined calendar attributes:

- **Server Version**

This read-only attribute shows the version number of the server managing the calendar. This attribute is a **CSA_VALUE_UINT32** type of attribute.

- **Data Version**

This read-only attribute shows the data version of the calendar. This attribute is a **CSA_VALUE_UINT32** type of attribute.

Entry Attributes

The following table lists the entry attributes supported in the CDE implementation. See the relevant man page for more information. The list of entry attributes is extensible through the extended naming convention.

CSA Entry Attributes

Attribute Name	Description	Read-Only
Audio Reminder	Audio reminder attribute	No
Classification	Classification attribute	No
Date Completed	Date completed attribute	No
Date Created	Entry creation date attribute	Yes
Description	Description attribute	No
Due Date	Due date attribute	No
End Date	End date attribute	No
Exception Dates	Exception dates attribute	No
Flashing Reminder	Flashing reminder attribute	No
Last Update	Last update attribute	Yes
Mail Reminder	Mail reminder attribute	No
Number Recurrences	Number of recurrences attribute (computed)	Yes

CSA Entry Attributes		
Attribute Name	Description	Read-Only
Organizer	Organizer attribute	Yes
Popup Reminder	Pop-up reminder attribute	No
Priority	Priority attribute	No
Recurrence Rule	Recurrence rule attribute	No
Reference Identifier	Reference identifier attribute	Yes
Sponsor	Sponsor attribute	No
Start Date	Start date attribute	No
Status	Status attribute	No
Subtype	Subtype attribute	No
Summary	Summary attribute	No
Transparency	Time transparency or blocking attribute	No
Type	Type attribute	Yes

The following section provides additional information about the entry attributes listed in the CSA Entry Attributes table.

- **Organizer**

The organizer of an entry is set to the user who is running the application that calls **csa_add_entry()** to add the entry to the calendar. It becomes read-only and cannot be changed after the entry is added.

- **Reference Identifier**

The reference identifier of an entry is a string that contains a unique identifier of the entry within the calendar as well as the name and location of the calendar. The format is **n:calendar@location** where **n** is a number that uniquely identifies the entry within the calendar, **calendar** is the name of the calendar, and **location** is the name of the machine where the calendar is stored.

- **Status**

The CDE implementation defines the following additional status values:

CSA_X_DT_STATUS_ACTIVE

CSA_X_DT_STATUS_DELETE_PENDING

CSA_X_DT_STATUS_ADD_PENDING

CSA_X_DT_STATUS_COMMITTED

CSA_X_DT_STATUS_CANCELLED

- **Type**

The value becomes read-only and cannot be changed after the entry is added. The CDE implementation defines the following additional type value:

CSA_X_DT_TYPE_OTHER

CDE Entry Attributes

The following are CDE-defined entry attributes:

- **Show Time**

The value of this attribute indicates whether the start and end time of the entry should be shown to the user. It can be modified using **csa_update_entry_attributes()**. This attribute is a **CSA_VALUE_SINT32** type of attribute.

- **Repeat Type**

The frequency of recurrence of the entry, which indicates how often the entry repeats. This is a read-only attribute and is derived from the entry attribute **Recurrence Rule**.

This attribute is a **CSA_VALUE_UINT32** type of attribute.

The following values are defined:

CSA_X_DT_REPEAT_ONETIME

CSA_X_DT_REPEAT_DAILY

CSA_X_DT_REPEAT_WEEKLY

CSA_X_DT_REPEAT_BIWEEKLY

CSA_X_DT_REPEAT_MONTHLY_BY_WEEKDAY

CSA_X_DT_REPEAT_MONTHLY_BY_DATE

CSA_X_DT_REPEAT_YEARLY

CSA_X_DT_REPEAT_EVERY_NDAY

CSA_X_DT_REPEAT_EVERY_NWEEK

CSA_X_DT_REPEAT_EVERY_NMONTH

CSA_X_DT_REPEAT_MON_TO_FRI

CSA_X_DT_REPEAT_MONWEDFRI

CSA_X_DT_REPEAT_TUETHUR

CSA_X_DT_REPEAT_WEEKDAYCOMBO

CSA_X_DT_REPEAT_OTHER

CSA_X_DT_REPEAT_OTHER_WEEKLY

CSA_X_DT_REPEAT_OTHER_MONTHLY

CSA_X_DT_REPEAT_OTHER_YEARLY

- **Repeat Times**

This attribute shows the number of times an entry repeats. This is a read-only attribute and is derived from the entry attribute **Recurrence Rule**. This attribute is a **CSA_VALUE_UINT32** type of attribute.

- **Repeat Interval**

This attribute tells how often an entry with repeat types **CSA_X_DT_REPEAT_EVERY_NDAY, CSA_X_DT_REPEAT_EVERY_NWEEK**, or **CSA_X_DT_REPEAT_EVERY_NMONTH** repeats. This is a read-only attribute, and is derived from the entry attribute **Recurrence Rule.** For example, if the value of this attribute is 3 and the repeat type is

CSA_X_DT_REPEAT_EVERY_NWEEK, the entry repeats every three weeks. This attribute is a **CSA_VALUE_UINT32** type of attribute.

- **Repeat Occurrence Number**

If the entry's repeat type is **CSA_X_DT_REPEAT_MONTHLY_BY_WEEKDAY**, this attribute tells in which week the entry repeats. This is a read-only attribute and is derived from the entry attribute **Recurrence Rule**. This attribute is a **CSA_VALUE_SINT32** type of attribute.

- **Sequence End Date**

This entry attribute shows the end date of the sequence. This is a read-only attribute and is derived from the entry attribute **Recurrence Rule**. This attribute is a **CSA_VALUE_DATE_TIME** type of attribute.

General Information about Functions

The following general information applies to all functions:

- Character set restriction

The calendar attribute, **CSA_CAL_ATTR_CHARACTER_SET**, is used to store the locale information of the calendar.

Note: All data except textual description passed in the library must be in ASCII format and the library supports single-byte as well as multibyte character strings.

- Type checking for attribute values is provided for the predefined attributes only.
- When a function takes both a session handle and an entry handle, the session handle is always ignored in the CDE implementation.
- Entry attributes **CSA_ENTRY_ATTR_RECURRENCE_RULE** and **CSA_ENTRY_ATTR_EXCEPTION_DATES** are used to specify recurrence information of a calendar entry. Information in the **CSA_ENTRY_ATTR_RECURRENCE_RULE** attribute can be queried using the following attributes: **CSA_X_DT_ENTRY_ATTR_REPEAT_TYPE**, **CSA_X_DT_ENTRY_ATTR_REPEAT_TIMES**, **CSA_X_DT_ENTRY_ATTR_REPEAT_INTERVAL**, **CSA_X_DT_ENTRY_ATTR_REPEAT_OCCURRENCE_NUM**, and **CSA_X_DT_ENTRY_ATTR_SEQUENCE_END_DATE**. These computed attributes are read-only.
- The **CSA_calendar_user** data structure specifies either a user or a calendar. In the first case, for example, when specifying a user in an access list, only the **user_name** field is used and all other fields are ignored. In the latter case, for example, when specifying the calendar to log onto, only the **calendar_address** field is used and all other fields are ignored. The format is **calendar@location** where **calendar** is the name of the calendar and **location** is the name of the machine where the calendar is stored.
- Attributes of value type **CSA_VALUE_ATTENDEE_LIST** are not supported and **CSA_E_INVALID_ATTRIBUTE_VALUE** will be returned if they are specified.
- Although the **repeat_count** and **snooze_time** fields in the **CSA_reminder** data structure are stored in the calendar, the calendar service does not interpret their values and the associated reminder will be returned only once by the server.
- The user interface extension **CSA_X_UI_ID_EXT** is not supported.

Administration Functions

This section contains descriptions for the administration functions supported in the CDE implementation. See the relevant man page for more information.

- Free - Frees memory allocated by the calendaring service.

```
CSA_return_code
csa_free(
        CSA_buffer              memory
);
```

- List Calendars - Lists the calendars supported by a calendar server.

```
CSA_return_code
csa_list_calendars(
                CSA_service_reference  calendar_service,
                CSA_uint32             *number_names,
                CSA_calendar_user      **calendar_names,
                CSA_extension          *list_calendars_extensions
);
```

A host name where the server runs should be passed in **calendar_server**.

- Logon – Logs on to the calendar service and establishes a session with a calendar.

```
CSA_return_code

csa_logon(
              CSA_service_reference      calendar_service,
            CSA_calendar_user       *user,
            CSA_string              password,
            CSA_string              character_set,
            CSA_string              required_csa_version,
            CSA_session_handle      *session,
            CSA_extension          *logon_extensions
);
```

Arguments **calendar_service, password, character_set,** and **required_csa_version** are not used.

The **calendar_address** field of the **CSA_calendar_user** structure pointed to by **user** specifies the calendar to log onto. The format is **calendar@location** where **calendar** is the name of the calendar and **location** is the host name where the calendar is stored.

The CDE-defined extension **CSA_X_DT_GET_USER_ACCESS_EXT** is supported. This extension can be used to get the access rights the calling user has with respect to the calendar. The user's access rights is returned in the **item_data** field of the extension structure.

- Logoff – Terminates a session with a calendar.

```
CSA_return_code

csa_logoff(

         CSA_session_handle          session,

          CSA-extension               *logoff_extensions

);
```

- Query Configuration – Determines information about the installed CSA configuration.

CSA_return_code

```
csa_query_configuration(
        CSA_session_handle      session,
        CSA_enum                item,
        CSA_buffer              *reference,
        CSA_extension           *query_configuration_extensions
);
```

The following items are not supported by this implementation of CDE:

CSA_CONFIG_CHARACTER_SET

CSA_CONFIG_LINE_TERM

CSA_CONFIG_VER_IMPLEM

Calendar Management Functions

This section contains descriptions for the calendar management functions supported in the CDE implementation. See the relevant man page for more information.

- Add calendar – Adds a calendar to the calendar service.

```
CSA_return_code
csa_add_calendar(
        CSA_session_handle      session,
        CSA_calendar_user       *user,
        CSA_uint32              number_attributes,
        CSA_attribute           *calendar_attributes,
        CSA_extension           *add_calendar_extensions
);
```

  The first argument **session** is ignored.

  The **calendar_address** field of the **CSA_calendar_user** structure pointed to by **user** specifies the name and the location of the calendar to be created. The format is **calendar@location** where **calendar** is the name of the calendar and **location** is the host name where the calendar is to be stored; for example, **my_calendar@my_host**.

- Call Callbacks – Forces the invocation of the callback functions associated with the specified callback list(s).

```
CSA_return_code
csa_call_callbacks(
       CSA_session_handle              session,
       CSA_flags                       reason,
       CSA_extension                   *call_callbacks_extensions
);
```

- Delete Calendar – Deletes a calendar from the calendar service.

```
CSA_return_code
csa_delete_calendar
       CSA_session_handle             session,
       csa_extension                  *delete_calendar_extensions
);
```

- List Calendar Attributes – Lists the names of the calendar attributes associated with a calendar.

```
CSA_return_code
csa_list_calendar_attributes(
        CSA_session_handle         session,
        CSA_uint32                 *number_names,
        CSA_attribute_reference    **calendar_attributes_names,
        CSA_extension       *list_calendar_attributes_extensions
);
```

- Read Calendar Attributes – Reads and returns the calendar attribute values for a calendar.

```
CSA_return_code
csa_read_calendar_attributes(
              CSA_session_handle      session,
              CSA_uint32              number_names,
              CSA_attribute_reference *attribute_names,
              CSA_uint32              *number_attributes,
              CSA_attribute           **calendar_attributes,
              CSA_extension  *read_calendar_attributes_extensions
);
```

- Register Callback Functions – Registers the callback function to be invoked with the specified type of updates in the calendar.

```
CSA_return_code
csa_register_callback(
            CSA_session_handle       session,
            CSA_flags                reason,
            CSA_callback             callback,
            CSA_buffer               client_data,
            CSA_extension            register_callback_extensions
);
```

- Unregister Callback Functions – Unregisters the specified callback function.

```
CSA_return_code
csa_unregister_callback(
            CSA_session_handle    session,
            CSA_flags             reason,
            CSA_callback          callback,
            CSA_buffer            client_data,
            CSA_extension         *unregister_callback_extensions
);
```

- Update Calendar Attributes – Updates the calendar attribute values for a calendar.

```
CSA_return_code
csa_update_calendar_attributes(
        CSA_session_handle        session,
        CSA_uint32                number_attributes,
        CSA_attribute             *calendar_attributes,
        CSA_extension  *update_calendar_attributes_extensions
);
```

Entry Management Functions

This section contains descriptions for the entry management functions supported in the CDE implementation. See the relevant man page for more information.

- Add Entry – Adds an entry to the specified calendar.

```
CSA_return_code
csa_add_entry(
      CSA_session_handle    session,
      CSA_uint32            number_attributes,
      CSA_attribute         *entry_attributes,
      CSA_entry_handle      *entry,
      CSA_extension         *add_entry_extensions
);
```

- Delete Entry – Deletes an entry from the specified calendar.

```
CSA_return_code
csa_delete_entry(
        CSA_session_handle      session,
        CSA_entry_handle        entry,
        CSA_enum                delete_scope,
        CSA_extension           *delete_entry_extensions
);
```

- List Entries – Lists the calendar entries that match all the attribute search criteria.

```
CSA_return_code
csa_list_entries(
        CSA_session_handle      session,
        CSA_uint32              number_attributes,
        CSA_attribute           *entry_attributes,
        CSA_enum                *list_operators,
        CSA_uint32              *number_entries,
        CSA_entry_handle        **entries,
        CSA_extension           *list_entries_extensions
);
```

  The following information details more about the operators specified in **list_operators**:

  Only the operators **CSA_MATCH_ANY** and **CSA_MATCH_EQUAL_TO** are supported for the attribute value types **CSA_VALUE_REMINDER**, **CSA_VALUE_CALENDAR_USER**, and **CSA_VALUE_DATE_TIME_RANGE**.

  Only the operators **CSA_MATCH_ANY**, **CSA_MATCH_EQUAL_TO**, **CSA_MATCH_NOT_EQUAL_TO**, and **CSA_MATCH_CONTAIN** are supported for the attribute value type **CSA_VALUE_STRING**. The operator **CSA_MATCH_CONTAIN** only applies to **CSA_VALUE_STRING** type of attributes.

  Matching of attributes with the value types **CSA_VALUE_OPAQUE_DATA**, **CSA_VALUE_ACCESS_LIST, CSA_VALUE_ATTENDEE_LIST,** and, **CSA_VALUE_DATE_TIME_LIST** are not supported. The only exception is the attribute **CSA_ENTRY_ATTR_REFERENCE_IDENTIFIER**. The operator **CSA_MATCH_EQUAL_TO** is supported for this attribute.

- List Entry Attributes – Lists the names of the entry attributes associated with the specified entry.

```
CSA_return_code

csa_list_entry_attributes(
      CSA_session_handle       session,
      CSA_entry_handle         entry,
      CSA_uint32               *number_names,
      CSA_attribute_reference  **entry_attribute_names,
      CSA_extension            *list_entry_attributes_extensions
);
```

- List Entry Sequence – Lists the recurring calendar entries that are associated with a calendar entry.

```
CSA_return_code
csa_list_entry_sequence(
     CSA_session_handle  session,
     CSA_entry_handle    entry,
     CSA_date_time_range time_range,
     CSA_uint32          *number_entries,
     CSA_entry_handle    **entry_list,
     CSA_extension       *list_entry_sequences_extensions
);
```

CSA_E_INVALID_PARAMETER is returned if the specified entry is a one-time entry.

- Read Entry Attributes – Reads and returns the calendar entry attribute values for a specified entry.

```
CSA_return_code
csa_read_entry_attributes(
       CSA_session_handle      session,
       CSA_entry_handle        entry,
       CSA_uint32              number_names,
       CSA_attribute_reference *attribute_names,
       CSA_uint32              *number_attributes,
       CSA_attribute           **entry_attributes,
       CSA_extension           *read_entry_attributes_extensions
);
```

- Read Next Reminder – Reads the next reminder of the given type in the specified calendar relative to a given time.

```
CSA_return_code
csa_read_next_reminder(
     CSA_session_handle      session,
     CSA_uint32              number_names,
     CSA_attribute_reference *reminder_names,
    CSA_date_time           given_time,
    CSA_uint32              *number_reminders,
    CSA_remainder_reference **reminder_references,
    CSA_extension   *read_next_reminder_extensions
);
```

- Update Entry Attributes – Updates the calendar entry attributes.

```
CSA_return_code
csa_update_entry_attributes(
        CSA_session_handle  session,
        CSA_entry_handle    entry,
        CSA_enum            update_scope,
        CSA_boolean         update_propagation,
        CSA_uint32          number_attributes,
        CSA_attribute       *entry_attributes,
        CSA_entry_handle    *new_entry,
        CSA_extension       *update_entry_attributes_extensions
);
```

Update propagation is not supported; the **update_propagation** argument should be set to **CSA_FALSE.**

Glossary

action A user interface defined in a database of files that requests an application perform some operation.

action icon An icon that represents an action in a File Manager or Application Manager window. It is displayed by creating an executable file with the same name as the action it represents.

action server A host computer that provides access to a collection of actions.

active A window, window element, or icon that is currently affected by keyboard and mouse input. Active windows are differentiated from other windows on the workspace by a distinctive title bar color or shade. An active window element is indicated by a highlight or selection cursor.

app–defaults file
A file for each application that programmers use to define the X resources.

application group
An Application Manager container that holds a specific software application.

Application Manager
The software application that manages the tools and other software applications available to you.

application server
A host computer that provdes access to a software application.

argument An item of information following a command.

arrow keys The four directional keys on a keyboard. Also see navigation keys.

attachment An encapsulated data object inside a document.

background The underlying area of a window on which objects, such as buttons and lists, are displayed.

bitmap An image stored in a raster format. Usually refers to an image limited to two colors (a foreground and a background color). Contrast with pixmap.

bitmapped font
A font made from a matrix of dots. See font.

busy pointer The mouse pointer displayed when an application is busy and cannot accept input.

button A generic term for a window control that initiates an action by an application, usually executing a command, displaying a window, or displaying a menu. Also used to describe the controls on a mouse.

button binding Association of a mouse button operation with a particular behavior.

cascaded list The list box that displays additional elements from which you choose in order to interact with other screen elements.

cascaded menu
The menu item that displays additional elements from which you choose in order to interact with other screen elements.

CDE Common Desktop Environment, a graphical user interface running on UNIX.

check box A nonexclusive control whose setting is indicated by the presence or absence of a check mark. A check box has two states, on and off.

choose To use the mouse or keyboard to pick a menu command, button, or icon that begins a command or action. Contrast with select.

click To press and release a mouse button without moving the mouse pointer.

client A system or software application that requests services from a network server.

clipboard A buffer that temporarily stores the last cut, copy, or paste data or object.

combo box Use text box to refer to the text box portion of a combo box, and list box to refer to the list portion. Example: Type the name of the file in the File text box or select it from the list box underneath.

command line prompt
A prompt, usually %, >, or $, that shows the computer is ready to accept commands. In a terminal emulation window, you display the command line prompt by pressing Return.

context–sensitive help
Help information about the specific choice or object that the cursor or pointer is on.

control A generic term for a variety of objects (such as buttons, check boxes, and scroll bars) that perform an action or indicate an option setting. Also applies to Front Panel icons.

Copy View Options
A menu command that copies the current view's properties and places the copy onto the clipboard.

current item The currently highlighted item in a list.

current session
The session saved by Session Manager when you log off. At the next login, unless you specify otherwise, this session automatically opens, enabling work to continue where you left off. Contrast with home session.

current setting The present state of a control such as a check box or radio button.

current workspace
The workspace that is presently displayed on the screen. It can be changed with the workspace switch.

cursor A graphical device that indicates the current object that will be affected by mouse or keyboard input.

data host A host computer where the data for an action is located.

DATA_HOST An attribute added to the DATA_ATTRIBUTES entry that indicates the host system that the Data Type was loaded from. Note that this value should not be set in the *.dt files but is generated when the database is loaded.

database host A host computer where an action is defined.

data–type A mechanism that associates particular data files with the appropriate applications and actions. Data types can determine the type of a file based on file–naming conventions, such as a particular extension name, or on the contents of the file.

default A value set automatically by an application.

dialog box A window displayed by an application that requires user input. Do not use dialog as shorthand.

directory A collection of files and other subdirectories.

display–dependent session
A session that can be restored on only a particular display.

display–independent session
A session that can be restored on any display, regardless of screen resolution or color capability.

double–click To quickly press a mouse button twice without moving the mouse pointer. Unless otherwise specified, button 1 is assumed.

drag To move the mouse pointer over an object, press and hold down mouse button 1, and then move the mouse pointer and the object to another location on the workspace.

drag and drop To directly manipulate an object by using a pointing device to move and place the object somewhere else.

drag over feedback
The drag icon changes appearance when the user drags it over potential drop zones.

drag under feedback
The appearance provided by a drop zone. The feedback can be a solid line drawn around the site, a raised or lowered surface with a beveled edge around the drop zone, or a pixmap drawing over the drop zone.

drop After grabbing an object, the act of releasing the mouse button. If the object is dropped in an appropriate area, an Action is initiated. Also see grab.

drop target A rectangular graphic that represents the drop zone in an application.

drop zone An area of the workspace, including the Trash, Printer, and CDE Mail icons, that accepts a dropped object. Objects can be dropped on the workspace for quick access.

element A generic term for any entity that can be considered a standalone item in a broader context, such as an item in a list or a control in a window.

execution host A host computer where an application invoked by an action runs. This may be the same computer where the action resides, or it may be another computer on the network.

field A window element that holds data, as in the Name field or the Telephone number field. Preferred: Use a more specific noun to describe the element, as in the Name text box or the Files list box.

File Manager The software application that manages the files and directories on your system.

file server A host computer that stores data files used by applications.

format type In CDE document containers, the type used to store the property.

focus Place to which keyboard input is directed.

folder An icon that represents a directory.

font A complete set of characters (letters, digits, and special characters) of one size and one typeface. Ten–point Helvetica bold is an example of a font.

foreground The content of a window and the color or shading used to distinguish it from the window's background.

Front Panel A centrally located window containing icons for accessing applications and utilities, including the workspace switch. The Front Panel occupies all workspaces.

grab To move the mouse pointer over an object, and then to press and hold down mouse button 1 in preparation for moving the object. Also see drag; drop.

grab handles (or handles) The small squares displayed at the corners and midpoints of a selected graphic object.

group box A box in a window that visually associates a set of controls.

home directory A directory where you keep personal files and additional directories. By default, File Manager and Terminal Emulator windows are set to the home directory when you first open them.

home session A choice at logout to designate a particular session, other than the one you are currently in, as the one you will automatically return to at the next login.

hyperlink In Help text, information that when chosen displays another Help topic.

icon A graphic symbol displayed on the screen, which you can select to work in a particular function or software application.

insertion point The point at which data typed on the keyboard, or pasted from the clipboard or a file, appears on the screen. In text, a synonym for cursor.

Install Icon A choice on subpanels that enables you to install icons on the desktop using drag and drop.

IS_ACTION An attribute added to the DATA_ATTRIBUTES entry that is created when an action is loaded. The DtDtsDataTypeIsAction uses this attribute to determine if a data type was created from the action table. It has no representation in the *.dt files except as an action entry. Note that this value should not be set in the *.dt files and is only used internally.

IS_SYNTHETIC An attribute added to the DATA_CRITERIA/DATA_ATTRIBUTES entries that is created when an action is loaded. It has no representation in the *.dt files except as an action entry. Note: This value should not be set in the *.dt files and is only used internally.

ITE Internal Terminal Emulator. ITE allows use of a bitmapped display as a terminal (through command–line mode from the Login screen).

items Elements in a list.

key binding Association of a keystroke with a particular behavior.

label The text appearing next to a window element that names the element.

link Synonym for symbolic link.

list A control that contains elements from which you select. Also called selection list.

list box Any of a number of graphical devices that displays a list of items from which you can select one or more items. It is usually not necessary to name the specific kind of box being used.

local host The CPU or computer on which a software application is running; your workstation.

mapping An action that invokes another action rather than containing its own EXEC–STRING. The file /usr/vue/types/user–prefs.vf contains the built–in mapped actions. For example, the built–in CDE Mail action used by the Front Panel is mapped to the Elm action.

menu A list of commands from which you select to perform a particular application task.

menu bar The part of the application window between the title bar and the work area where menu names are listed.

menu item A choice that appears on a menu.

metadata In Bento containers, the information about an object. Contrast with value.

minimize To turn a window into an icon. The push button that minimizes a window is located near the upper–right corner of the window frame.

modifier key A key that when pressed and held along with another key changes the meaning of the second key. Control, Alt, and Shift are examples.

multipart document
A document that contains one or more attachments.

navigation keys
The keyboard keys used to move the current location of the cursor. These include the arrow keys (with or without the Control key); the Tab key (with or without the Control or Shift keys); the Begin and End keys (with or without the Control key); and the Page Up and Page Down keys.

networked session
A session managed across multiple systems. Using a networked session enables the same session to be seen, regardless of which system was used to log in. It also provides a single home directory across multiple systems.

newline character
An unseen character that marks the end of a line of text in a document. It tells a printer or screen to break a line and start a new one.

no saveback The inability of drag and drop to write changes in data held in buffers back to an originating file.

On Item help A form of help in which an application provides on–screen information about a particular command, operation, dialog box, or control.

operation indicator
The part of a drag icon that gives users feedback on the operation (move, copy, or link) that is occurring during the drag.

options A generic term that applies to the variations available when choosing a command to run, or when selecting or filling in items in a dialog box.

page To advance text displayed in a window by one full screen at a time, usually using a scroll bar.

palette The range of available elements, usually colors.

pixmap An image stored in a raster format. Usually refers to an image that may have more than two colors. Contrast with bitmap.

point To move the mouse until the pointer rests on a particular screen object or area.

pointer The arrow or other graphical marker that indicates the current mouse position, and possibly the active window. Also see cursor.

print server A host computer to which one or more printers are connected, or the UNIX process that manages those printers.

Properties A menu command that enables you to set characteristics of an object, such as its date or name, or display identifying characteristics of an object, such as typefaces.

push button A control that immediately starts an action as soon as it is chosen. OK, Cancel, and Help are examples of push buttons commonly found in dialog boxes.

radio button An exclusive control whose setting is indicated by the presence or absence of a graphical indicator, usually part of a radio group. A radio button has two states, on and off.

radio button group
A box containing a set of radio buttons that may have a distinct label. At most, one of the radio buttons may be activated at a time.

release To let go of a mouse button or keyboard key.

resize handle A control used to change the size of a window or a pane in a window.

resize pointer The mouse pointer displayed when an object, such as a window, is being resized.

resource A mechanism of the X Window System for specifying an attribute (appearance or behavior) of a window or application. Resources are usually named after the elements they control.

saveback The ability of drag and drop to write modified data back to the originating file.

scalable typeface
A mathematical outline for a typeface used to create a bitmapped font for a particular size, slant, or weight.

screen lock A function that locks the workstation screen, barring further input until the valid user password is entered.

screen saver A choice that causes the workstation, after a specified time period, to switch off the display or to vary the images that are displayed, thereby prolonging the life of the screen.

scroll bar A control located at the right or bottom of a window that enables you to display window content not currently visible.

select To add highlighting or some other visual cue to an object so that it can be operated or enabled. Selection does not imply the initiation of an action but

rather a change of state, such as highlighting an item in a list, or toggling a check box on.

server A system that supplies services to a client.

session The elapsed time between user login and logout.

session server A system that provides networked sessions. Session files reside on the session server and are used whenever you log into a system on the network.

shortcut General term for a mouse action that simplifies filling out a dialog box. For example: As a shortcut, double–click an item in the Filename list box to select it and choose OK in one action.

shortcut keys A keyboard key sequence used to activate a menu command. This can be a key sequence that uses a special accelerator key, or an underlined letter (mnemonic) sequence. For example: Press Alt+F4 or Alt+F+P to choose the command File⇒Πριντ.

slider A control that uses a track and arm to set a value from among the available values. The position of the arm (or a separate indicator) gives the currently set value.

software application
A computer program that provides you with tools to do work. Style Manager, Text Editor, and File Manager are examples of software applications.

source indicator
The part of a drag icon that represents the item being dragged.

spin box A window element with a text box and two arrow buttons that displays a set of related but mutually exclusive choices, such as days of the week.

state indicator The part of a drag icon that is used as a pointer for positioning combined with feedback that shows whether a drop zone is valid or invalid.

Style Manager The software application used to customize some of the visual elements and system device behaviors of the workspace environment, including colors and fonts, and keyboard, mouse, window, and session start–up behaviors.

subpanel A component of the Front Panel that provides additional controls. Subpanels usually contain groups of related controls.

symbolic link A reference to a file or directory.

synthetic data attribute
A data type added to the DATA_CRITERIA/DATA_ATTRIBUTES entries that is created when an action is loaded. It has no representation in the ***.dt** filesexcept as an action entry. Note that this attribute should not be set in the ***.dt** files and is only used internally.

terminal emulator
A window that emulates a particular type of terminal for running nonwindow programs. Terminal emulator windows are most commonly used for typing commands to interact with the computer’s operating system.

text field A rectangular area in a window where information is typed. Text fields with keyboard focus have a blinking text insertion cursor.

tile A rectangular area used to cover a surface with a pattern or visual texture. For example: The Workspace Manager supports tiling, enabling users with limited system color availability to create new color tiles blended from existing colors.

title bar The topmost area of a window, containing the window title.

toggle To change the state of a two–state control, such as a radio button or check box, using either the mouse or keyboard.

value In Bento containers, refers to the contents of an object. Contrast with metadata.

window A rectangular area on the display. Software applications typically have one main window from which secondary windows, called dialog boxes, can be opened.

window frame The visible part of a window that surrounds a software application. A window frame can contain up to five controls: title bar, resize borders, minimize button, maximize button, and the Window menu button.

window icon A minimized window.

Window list An action that presents a list of all the open windows associated with the window from which the action was selected.

Window menu The menu displayed by choosing the Window menu button. The menu provides choices that manipulate the location or size of the window, such as Move, Size, Minimize, and Maximize.

Window menu button
The control at the upper–left corner of a window, next to the title bar. Choosing it displays the Window menu.

work area The part of a window where controls and text appear.

workspace The current screen display, the icons and windows it contains, and the unoccupied screen area where objects can be placed.

workspace background
The portion of the display not covered by windows, icons, or objects.

Workspace Manager
The software application that controls the size, placement, and operation of windows within multiple workspaces. The Workspace Manager includes the Front Panel, the window frames that surround each application, and the Window and Workspace menus.

Workspace menu
The menu displayed by pointing at an unoccupied area of the workspace and clicking mouse button 3.

workspace object
An object that has been copied from File Manager to the workspace.

workspace switch
A control that enables you to select one workspace from among several workspaces.

E

F

H

I

L

M

N

O

P

R

S

T

U

V

W

X