

ULTRIX

Worksystem Software

digital

Guide to the XUI Toolkit Intrinsic:
C Language Binding

Order Number: AA-MA96A-TE

**ULTRIX Worksystem Software
Guide to the XUI Toolkit Intrinsic:
C Language Binding**

ULTRIX Worksystem Software, Version 2.0

Digital Equipment Corporation

Copyright © 1988 Digital Equipment Corporation
All rights reserved.


Copyright © 1984, 1985, 1986, 1988 Massachusetts Institute of Technology, Cambridge, Massachusetts.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

DEC	ULTRIX	VMS
DECnet	ULTRIX-11	VT
DECUS	ULTRIX-32	XUI
DECwindows	VAX	ULTRIX Worksystem Software
MicroVAX	VAXstation	

UNIX is a registered trademark of AT&T in the USA and other countries.

X Window System is a trademark of MIT.

This manual is derived from MIT documentation, which contains the following permission notice: Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of MIT or DIGITAL not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. MIT and DIGITAL make no representations about the suitability of the software described herein for any purpose. It is provided "as is," without express or implied warranty.

This manual was written and produced by the ULTRIX Documentation Group in Nashua, New Hampshire.

Contents

About This Manual

Audience	xi
Organization	xi
Related Documents	xiii
Conventions	xiv

1 Intrinsic and Widgets

1.1 Terminology	1-2
1.2 Intrinsic	1-3
1.3 Widgets	1-5
1.3.1 Core Widgets	1-6
1.3.1.1 CoreClassPart Structure	1-6
1.3.1.2 CorePart Structure	1-7
1.3.1.3 CorePart Default Values	1-8
1.3.2 Composite Widgets	1-9
1.3.2.1 CompositeClassPart Structure	1-9
1.3.2.2 CompositePart Structure	1-10
1.3.2.3 CompositePart Default Values	1-10
1.3.3 Constraint Widgets	1-11
1.3.3.1 ConstraintClassPart Structure	1-11
1.3.3.2 ConstraintPart Structure	1-12

1.4 Widget Classing	1-12
1.4.1 Widget Naming Conventions	1-13
1.4.2 Widget Subclassing in Public .h Files	1-14
1.4.3 Widget Subclassing in Private .h Files	1-15
1.4.4 Widget Subclassing in .c Files	1-17
1.4.5 Widget Class and Superclass Lookup	1-20
1.4.6 Widget Subclass Verification	1-20
1.4.7 Superclass Chaining	1-21
1.4.8 Class Initialization: class_initialize and class_part_initialize Procedures	1-22
1.4.9 Inheritance of Superclass Operations	1-24
1.4.10 Invocation of Superclass Operations	1-25

2 Widget Instantiation

2.1 Initializing the XUI Toolkit	2-2
2.2 Loading the Resource Database	2-5
2.3 Parsing the Command Line	2-7
2.4 Creating Widgets	2-9
2.4.1 Creating and Merging Argument Lists	2-10
2.4.2 Creating a Widget Instance	2-11
2.4.3 Creating an Application Shell Instance	2-13
2.4.4 Widget Instance Initialization: the initialize Procedure	2-14
2.4.5 Constraint Widget Instance Initialization: the constraint_initialize Procedure	2-16
2.4.6 Nonwidget Data Initialization: the initialize_hook Procedure	2-16
2.5 Realizing Widgets	2-16
2.5.1 Widget Instance Window Creation: the realize Procedure	2-18
2.5.2 Window Creation Convenience Routine	2-19
2.6 Obtaining Window Information from a Widget	2-20
2.6.1 Unrealizing Widgets	2-21
2.7 Destroying Widgets	2-22

2.7.1 Adding and Removing Destroy Callbacks	2-23
2.7.2 Dynamic Data Deallocation: the destroy Procedure	2-24
2.7.3 Dynamic Constraint Data Deallocation: the constraint destroy Procedure	2-25
2.8 Exiting from an Application	2-25

3 Composite Widgets and Their Children

3.1 Verifying the Class of a Composite Widget	3-2
3.2 Addition of Children to a Composite Widget: the insert_child Procedure	3-2
3.3 Insertion Order of Children: the insert_position Procedure	3-3
3.4 Deletion of Children: the delete_child Procedure	3-4
3.5 Adding and Removing Children from the Managed Set	3-4
3.5.1 Managing Children	3-4
3.5.2 Unmanaging Children	3-6
3.5.3 Determining if a Widget Is Managed	3-7
3.6 Controlling When Widgets Get Mapped	3-7
3.7 Constrained Composite Widgets	3-8

4 Shell Widgets

4.1 Shell Widget Definitions	4-2
4.1.1 ShellClassPart Definitions	4-2
4.1.2 ShellPart Definition	4-5
4.1.3 ShellPart Default Values	4-7
4.1.4 Digital's Vendor Shell Implementation	4-9

5 Pop-Up Widgets

5.1 Pop-Up Widget Types	5-1
-------------------------------	-----

5.2 Creating a Pop-Up Shell	5-2
5.3 Creating Pop-Up Children	5-3
5.4 Mapping a Pop-Up Widget	5-3
5.5 Unmapping a Pop-Up Widget	5-6

6 Geometry Management

6.1 Initiating Geometry Changes	6-1
6.2 General Geometry Manager Requests	6-2
6.3 Resize Requests	6-5
6.4 Potential Geometry Changes	6-5
6.5 Child Geometry Management: the geometry_manager Procedure	6-6
6.6 Widget Placement and Sizing	6-8
6.7 Preferred Geometry	6-9
6.8 Size Change Management: the resize Procedure	6-11

7 Event Management

7.1 Adding and Deleting Additional Event Sources	7-1
7.1.1 Adding and Removing Input Sources	7-2
7.1.2 Adding and Removing Timeouts	7-3
7.2 Constraining Events to a Cascade of Widgets	7-4
7.3 Focusing Events on a Child	7-6
7.4 Querying Event Sources	7-7
7.5 Dispatching Events	7-9
7.6 The Application Input Loop	7-10
7.7 Setting and Checking the Sensitivity State of a Widget	7-10
7.8 Adding Background Work Procedures	7-11

7.9 X Event Filters	7-13
7.9.1 Pointer Motion Compression	7-13
7.9.2 Enter/Leave Compression	7-13
7.9.3 Exposure Compression	7-13
7.10 Widget Exposure and Visibility	7-14
7.10.1 Redisplay of a Widget: the expose Procedure	7-14
7.10.2 Widget Visibility	7-15
7.11 X Event Handlers	7-15
7.11.1 Event Handlers that Select Events	7-16
7.11.2 Event Handlers that Do Not Select Events	7-17
7.11.3 Current Event Mask	7-19

8 Callbacks

8.1 Using Callback Procedure and Callback List Definitions	8-1
8.2 Identifying Callback Lists	8-2
8.3 Adding Callback Procedures	8-2
8.4 Removing Callback Procedures	8-3
8.5 Executing Callback Procedures	8-4
8.6 Checking the Status of a Callback List	8-4

9 Resource Management

9.1 Resource Lists	9-1
9.2 Byte Offset Calculations	9-5
9.3 Superclass-to-Subclass Chaining of Resource Lists	9-6
9.4 Subresources	9-6
9.5 Obtaining Application Resources	9-7
9.6 Resource Conversions	9-8

9.6.1	Predefined Resource Converters	9-9
9.6.2	New Resource Converters	9-9
9.6.3	Issuing Conversion Warnings	9-12
9.6.4	Registering a New Resource Converter	9-12
9.6.5	Resource Converter Invocation	9-14
9.7	Reading and Writing Widget State	9-15
9.7.1	Obtaining Widget State	9-15
9.7.1.1	Widget Subpart Resource Data: the <code>get_values_hook</code> Procedure	9-16
9.7.1.2	Widget Subpart State	9-16
9.7.2	Setting Widget State	9-17
9.7.2.1	Widget State: the <code>set_values</code> Procedure	9-18
9.7.2.2	Widget State: the <code>set_values_almost</code> Procedure	9-20
9.7.2.3	Widget State: the <code>constraint set_values</code> Procedure	9-21
9.7.2.4	Widget Subpart State	9-21
9.7.2.5	Widget Subpart Resource Data: the <code>set_values_hook</code> Procedure	9-22

10 Translation Management

10.1	Action Tables	10-1
10.1.1	Action Table Registration	10-2
10.1.2	Action Names to Procedure Translations	10-3
10.2	Translation Tables	10-3
10.2.1	Event Sequences	10-4
10.2.2	Action Sequences	10-4
10.3	Translation Table Management	10-5
10.4	Using Accelerators	10-7
10.5	KeyCode-to-KeySym Conversions	10-9

11 Utility Functions

11.1 Determining the Number of Elements in an Array	11-1
11.2 Translating Strings to Widget Instances	11-1
11.3 Managing Memory Usage	11-2
11.4 Sharing Graphics Contexts	11-4
11.5 Managing Selections	11-5
11.5.1 Setting and Getting the Selection Timeout Value	11-6
11.5.2 Using Atomic Transfers	11-6
11.5.2.1 Atomic Transfer Procedures	11-7
11.5.2.2 Getting the Selection Value	11-9
11.5.2.3 Setting the Selection Owner	11-11
11.5.3 Using Incremental Transfers	11-12
11.5.3.1 Incremental Transfer Procedures	11-13
11.5.3.2 Getting the Selection Value	11-13
11.5.3.3 Setting the Selection Owner	11-19
11.6 Merging Exposure Events into a Region	11-20
11.7 Translating Widget Coordinates	11-21
11.8 Translating a Window to a Widget	11-21
11.9 Handling Errors	11-21

A Resource File Format

B Translation Table Syntax

Notation	B-1
Syntax	B-1
Modifier Names	B-2
Event Types	B-4
Canonical Representation	B-6
Examples	B-7

C Conversion Notes

D Standard Errors and Warnings

Error Messages D-1
Warning Messages D-3

E StringDefs.h Header File

Index

About This Manual

The *Guide to the XUI Toolkit Intrinsic: C Language Binding* describes the lower-level C functions that you can use to write XUI-based application programs (see related documents). Note that the information provided is specific to the C programming language.

Audience

The audience for this manual is the application programmer who will use one or more of the widget sets built with the Intrinsic and the widget programmers who will use the Intrinsic to build widgets for one of these widget sets. Not all the information in this manual, however, applies to both audiences. That is, the application programmer is likely to use only a number of the Intrinsic functions in writing an application, but the widget programmer is likely to use many more, if not all, of the Intrinsic functions in building a widget. Therefore, while the application programmer should be concerned with only sections of this manual, the widget programmer should be concerned with the whole manual. Note that the sections that are deemed to be of special interest to an applications programmer are identified in the next section.

This manual does not attempt to teach how to write an XUI application, nor does it attempt to teach C programming concepts.

Organization

The *Guide to the XUI Toolkit Intrinsic* contains the following:

Chapter 1 Intrinsic and Widgets

Provides a general overview of the XUI Toolkit. The application programmer should pay special attention to Sections 1.1, 1.2, 1.3, 1.4 and 1.4.1.

- Chapter 2** **Widget Instantiation**
Describes how to initialize the XUI toolkit, load the resource database, parse the command line, create a widget instance, realize a widget, obtain window information from a widget, destroy widgets, and exit an application. The application programmer should pay special attention to Sections 2.1, 2.2, 2.3, 2.4, 2.4.1, 2.4.2, 2.4.3, 2.5, 2.6, 2.6.1, 2.7, 2.7.1 and 2.8.
- Chapter 3** **Composite Widgets and Their Children**
Describes composite widgets and how to add or remove widgets from a managed set, manage or unmanage widgets, control when a widget gets mapped. The application programmer should pay special attention to Sections 3.1, 3.5, 3.5.1, 3.5.2, and 3.6.
- Chapter 4** **Shell Widgets**
Provides an overview to shell widgets.
- Chapter 5** **Pop-Up Widgets**
Describes pop-up widgets and how to create pop-up shells or widgets, and map or unmap pop-up widgets, The application programmer should pay special attention to Sections 5.2, 5.3, 5.4, and 5.5.
- Chapter 6** **Geometry Management**
Describes how to manage the geometry of your widgets.
- Chapter 7** **Event Management**
Describes how to use the XUI Toolkit event management mechanism. The application programmer should pay special attention to Sections 7.1, 7.1.1, 7.1.2, 7.2, 7.3 7.4, 7.5, 7.6 7.7, and 7.8.
- Chapter 8** **Callbacks**
Discusses callback procedures and how to use callback list definitions, add or remove callabacks, execute callbacks, and check the status of a callback list. The application programmer should pay special attention to Sections 8.1, 8.2, 8.3, and 8.4.
- Chapter 9** **Resource Management**
Describes to use the XUI Toolkit resource management mechanism. The application programmer should pay special attention to Sections 9.5, 9.6.4, 9.7, 9.7.1 and 9.7.2.

- Chapter 10 Translation Management
Describes to use the XUI Toolkit translation management mechanism. The application programmer should pay special attention to Sections 10.1, 10.3, and 10.4.
- Chapter 11 Utility Functions
Describes the XUI Toolkit utility functions that let you determine the number of elements in an array, translate strings to widget instances, manage memory usage, share graphics contexts, manipulate selections, merge exposure events into a region, translate widget coordinates, and translate a window to a widget. The application programmer should pay special attention to Sections 11.1, 11.2, 11.3, 11.4, 11.7, 11.8, and 11.9.
- Appendix A Resource File Format
Describes the format of the X Toolkit resource file.
- Appendix B Translation Table Syntax
Describes the syntax of a translation table.
- Appendix C Conversion Notes
Describes the functions that are provided to provide compatibility with earlier version of the Intrinsics.
- Appendix D Standard Errors and Warnings
Lists the XUI Toolkit error and warning messages.
- Appendix E StringDefs.h Header File
Lists the contents of the StringDefs.h header file.

Related Documents

XUI Style Guide

Describes the XUI user interface and, hence, the “look and feel” of an XUI application.

Guide to the XUI Toolkit: C Language Binding

Describes the widgets (user interface abstractions) that you can use to write your XUI-based application.

Guide to the Xlib Library: C Language Binding

Describes the low-level C functions that you can use to write your X-based application.

X Window System Protocol: X Version 11

Describes the precise semantics of the X11 protocol specification.

Conventions

The following typeface conventions are used in this manual:

- special** In text, all function names, events, errors, constant names, and pathnames are presented in this type.
- UPPERCASE** Although the ULTRIX system differentiates between lowercase and uppercase characters, uppercase is used intentionally in this manual where it is applicable.
- boldface** The primary occurrence for a given index entry is in this type.

In addition, the following conventions are used in this manual:

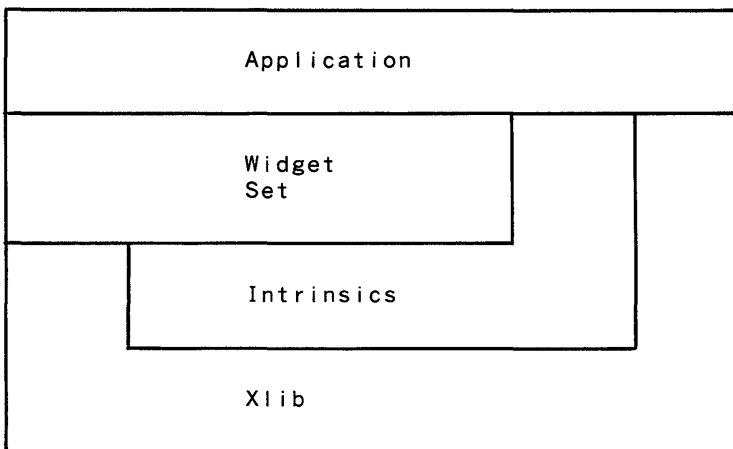
- Each function is introduced by a general discussion that distinguishes it from other functions. The function declaration itself follows, and each argument is specifically explained. General discussion of the function, if any is required, follows the arguments. see Section 8.12.2.
- To eliminate any ambiguity between those arguments that you pass and those that a function returns to you, the explanations for all arguments that you pass start with the word *specifies* or, in the case of multiple arguments, the word *specify*. The explanations for all arguments that are returned to you start with the word *returns* or, in the case of multiple arguments, the word *return*. The explanations for all arguments that you can pass and are returned start with the words *specifies and returns*.
- Any pointer to a structure that is used to return a value is designated as such by the `_return` suffix as part of its name. All other pointers passed to these functions are used for reading only. A few arguments use pointers to structures that are used for both input and output and are indicated by the `_in_out` suffix.

Intrinsics and Widgets 1

The Intrinsics and a widget set make up the XUI Toolkit. The Intrinsics provide the base mechanisms necessary to build a wide variety of widget sets and application environments. Because the Intrinsics mask implementation details from the widget and application programmer, the widgets and the application environments built with them are fully extensible and support independently developed new or extended components. By following a small set of conventions, widget programmers can extend their widget sets in new ways and can have these extensions function smoothly with the existing facilities.

The Intrinsics is a library package layered on top of Xlib. As such, the Intrinsics provide mechanisms (functions and structures) for extending the basic programming abstractions provided by the X Window System. By providing mechanisms for intercomponent and intracomponent interactions, the Intrinsics provide the next layer of functionality from which the widget sets are built.

Figure 1-1 illustrates this extended three-tiered X programming environment.



A typical XUI Toolkit application is most likely to be a client of a given widget set, a subset of the Intrinsics functions, and a smaller set of Xlib functions. This is illustrated by a left-to-right viewing of Figure 1-1. At the same time, a widget set is a client of both the Intrinsics and Xlib, and the Intrinsics are a client of Xlib only. This is illustrated by a top-to-bottom viewing of Figure 1-1.

For the application programmer, the XUI Toolkit provides:

- A consistent interface (widget set) for writing applications
- A small set of Intrinsics mechanisms that also are used in writing applications

For the widget programmer, the XUI Toolkit provides:

- A set of Intrinsics mechanisms for building widgets
- An architectural model for constructing and composing widgets
- A consistent interface (widget set) for programming

To the extent possible, the XUI Toolkit is policy free. The application environment, not the XUI Toolkit, defines, implements, and enforces:

- Policy
- Consistency
- Style

Each individual widget implementation defines its own policy. The XUI Toolkit design allows for the development of radically differing widget implementations.

1.1 Terminology

In addition to the terms already defined for X programming (see the *Guide to the Xlib Library*), the following terms are specific to the Intrinsics and are used throughout this book.

Application programmer

A programmer who uses the XUI Toolkit to produce an application user interface.

Class

The general group to which a specific object belongs.

Client

A function that uses a widget in an application or for composing other widgets.

Instance

A specific widget object as opposed to a general widget class.

Method

The functions or procedures that a widget class implements.

Name

The name that is specific to an instance of a widget for a given client.

Object

A software data abstraction consisting of private data and private and public functions that operate on the private data. Users of the abstraction can interact with the object only through calls to the object's public functions. In the XUI Toolkit, some of the object's public functions are called directly by the application, while others are called indirectly when the application calls the common Intrinsic functions. In general, if a function is common to all widgets, an application uses a single Intrinsic function to invoke the function for all types of widgets. If a function is unique to a single widget type, the widget exports the function as another "Xt" function.

Resource

A named piece of data in a widget that can be set by a client, by an application, or by user defaults.

User

A person interacting with a workstation.

Widget

An object providing a user-interface abstraction (for example, a Scrollbar widget).

Widget class

The general group to which a specific widget belongs, otherwise known as the type of the widget.

Widget programmer

A programmer who adds new widgets to the XUI Toolkit.

1.2 Intrinsic

The Intrinsic provide the base mechanisms (functions and structures) that simplify the design of application user interfaces. In addition, it assists widget and application programmers by providing a commonly used set of underlying user-interface functions to manage:

- Toolkit initialization
- Widgets

- Memory
- Window, file, and timer events
- Widget geometry
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmap
- Errors and warnings

Although all Intrinsic mechanisms are primarily intended for use by widget programmers, some are also intended for use by application programmers. The architectural model for the Intrinsic lets the widget programmer create new widgets by using the supplied mechanisms and/or by combining existing widgets. Therefore, an application interface layers built with the Intrinsic will provide a coordinated set of widgets and composition policies. While some of the widgets that are built with the Intrinsic are common across a number of application domains, others are restricted to a specific application domain.

The Intrinsic are based on an architectural model that also is flexible enough to accommodate a variety of different application interface layers. In addition, the supplied set of Intrinsic mechanisms are:

- Functionally complete and policy free
- Stylistically and functionally consistent with the X Window System primitives
- Portable across languages, computer architectures, and operating systems

Applications that use the Intrinsic mechanisms must include the following header files:

- `<X11/Intrinsic.h >`
- `<X11/StringDefs.h >`

In addition, they may also include:

- `<X11/Xatoms.h >`
- `<X11/Shell.h >`

Finally, widget implementations should include:

- `<X11/IntrinsicP.h >` instead of `<X11/Intrinsic.h >`

The applications should also include the additional headers for each widget class that they are to use (for example, `<X11/Label.h >` or `<X11/Scroll.h >`). On a UNIX-based system, the Intrinsics object library file is named `libXt.a` and is usually referenced as `-lXt`.

1.3 Widgets

The fundamental abstraction and data type of the XUI Toolkit is the *widget*, which is a combination of an X window and its associated semantics and which is dynamically allocated and contains state information. Logically, a widget is a rectangle with associated input/output semantics. Some widgets display information (for example, text or graphics), and others are merely containers for other widgets (for example, a menu box). Some widgets are output-only and do not react to pointer or keyboard input, and others change their display in response to input and can invoke functions that an application has attached to them.

Every widget belongs to exactly one widget class that is statically allocated and initialized and that contains the operations allowable on widgets of that class. Logically, a widget class is the procedures and data that is associated with all widgets belonging to that class. These procedures and data can be inherited by subclasses. Physically, a widget class is a pointer to a structure. The contents of this structure are constant for all widgets of the widget class but will vary from class to class. (Here, constant means the class structure is initialized at compile-time and never changed, except for a one-time class initialization and in-place compilation of resource lists, which takes place when the first widget of the class or subclass is created.) For further information, see Section 2.4.

The organization of the declarations and code for a new widget class between a public `.h` file, a private `.h` file, and the implementation `.c` file is described in Section 1.4. The predefined widget classes adhere to these conventions.

A widget instance is composed of two parts:

- A data structure that contains instance-specific values
- A class structure that contains information that is applicable to all widgets of that class

Much of the input/output of a widget (for example, fonts, colors, sizes, border widths, and so on) is customizable by users.

The next three sections discuss the base widget classes:

- Core widgets
- Composite widgets
- Constraint widgets

The chapter ends with a discussion of widget classing.

1.3.1 Core Widgets

The Core widget contains the definitions of fields common to all widgets. All widgets are subclasses of Core, which is defined by the CoreClassPart and CorePart structures.

1.3.1.1 CoreClassPart Structure

– The common fields for all widget classes are defined in the CoreClassPart structure:

```
typedef struct {
    WidgetClass superclass;           See Section 1.4
    String class_name;               See Section 1.4
    Cardinal widget_size;           See Section 2.4
    XtProc class_initialize;         See Section 1.4
    XtWidgetClassProc class_part_initialize; See Section 1.4
    Boolean class_inited;           See Section 1.4
    XtInitProc initialize;          See Section 2.4
    XtArgsProc initialize_hook;     See Section 2.4
    XtRealizeProc realize;           See Section 2.4
    XtActionList actions;           See Chapter 10
    Cardinal num_actions;           See Chapter 10
    XtResourceList resources;       See Chapter 9
    Cardinal num_resources;         See Chapter 9
    XrmClass xrm_class;             Private to resource manager
    Boolean compress_motion;        See Section 7.9.1
    Boolean compress_exposure;      See Section 7.9.3
    Boolean compress_enterleave;    See Section 7.9.2
    Boolean visible_interest;       See Section 7.10
    XtWidgetProc destroy;           See Section 2.7
    XtWidgetProc resize;           See Chapter 6
    XtExposeProc expose;            See Section 7.10
    XtSetValuesFunc set_values;     See Section 9.7
    XtArgsFunc set_values_hook;     See Section 9.7
    XtAlmostProc set_values_almost; See Section 9.7
    XtArgsProc get_values_hook;     See Section 9.7
    XtAcceptFocusProc accept_focus; See Section 7.3
    XtVersionType version;          See Section 1.4
    _XtOffsetList callback_private; Private to callbacks
    String tm_table;                See Chapter 10
    XtGeometryHandler query_geometry; See Chapter 6
}
```

```

        XtStringProc display_accelerator;           See Chapter 10
        caddr_t extension;                         See Section 1.4
    } CoreClassPart;

```

All widget classes have the core class fields as their first component. The prototypical `WidgetClass` is defined with only this set of fields. Various routines can cast widget class pointers, as needed, to specific widget class types, for example:

```

typedef struct {
    CoreClassPart core_class;
} WidgetClassRec, *WidgetClass;

```

The predefined class record and pointer for `WidgetClassRec` are:

```

extern WidgetClassRec widgetClassRec;
extern WidgetClass widgetClass;

```

The opaque types `Widget` and `WidgetClass` and the opaque variable `widgetClass` are defined for generic actions on widgets.

1.3.1.2 CorePart Structure

– The common fields for all widget instances are defined in the `CorePart` structure:

```

typedef struct _CorePart {
    Widget self;
    WidgetClass widget_class;           See Section 1.4
    Widget parent;                      See Section 1.4
    XrmName xrm_name;                  Private to resource manager
    Boolean being_destroyed;           See Section 2.7
    XtCallbackList destroy_callbacks;   See Section 2.7
    caddr_t constraints;               See Section 3.7
    Position x;                        See Chapter 6
    Position y;                        See Chapter 6
    Dimension width;                   See Chapter 6
    Dimension height;                  See Chapter 6
    Dimension border_width;           See Chapter 6
    Boolean managed;                   See Chapter 3
    Boolean sensitive;                 See Section 7.7
    Boolean ancestor_sensitive;        See Section 7.7
    XtEventTable event_table;         Private to event manager
    XtTmRec tm;                       Private to translation manager
    XtTranslations accelerators;       See Chapter 10
    Pixel border_pixel;               See Section 2.6

```

Pixmap border_pixmap;	See Section 2.6
WidgetList popup_list;	See Chapter 5
Cardinal num_popups;	See Chapter 5
String name;	See Chapter 9
Screen *screen;	See Section 2.6
Colormap colormap;	See Section 2.6
Window window;	See Section 2.6
Cardinal depth;	See Section 2.5
Pixel background_pixel;	See Section 2.6
Pixmap background_pixmap;	See Section 2.6
Boolean visible;	See Section 7.10
Boolean mapped_when_managed;	See Chapter 3

} CorePart;

All widget instances have the core fields as their first component. The prototypical type `Widget` is defined with only this set of fields. Various routines can cast widget pointers, as needed, to specific widget types; for example:

```
typedef struct {
    CorePart core;
} WidgetRec, *Widget;
```

1.3.1.3 CorePart Default Values

– The default values for the core fields, which are filled in by the Core resource list and the Core initialize procedure, are:

Field	Default Value
self	Address of the widget structure (may not be changed)
widget_class	widget_class argument to <code>XtCreateWidget</code> (may not be changed)
parent	parent argument to <code>XtCreateWidget</code> (may not be changed)
xrm_name	Encoded name argument to <code>XtCreateWidget</code> (may not be changed)
being_destroyed	Parent's being_destroyed value
destroy_callbacks	NULL
constraints	NULL
x	0
y	0
width	0
height	0

Field	Default Value
<code>border_width</code>	1
<code>managed</code>	False
<code>sensitive</code>	True
<code>ancestor_sensitive</code>	Bitwise AND of parent's <code>sensitive</code> & <code>ancestor_sensitive</code>
<code>event_table</code>	Initialized by the event manager
<code>tm</code>	Initialized by the translation manager
<code>accelerators</code>	NULL
<code>border_pixel</code>	XtDefaultForeground
<code>border_pixmap</code>	NULL
<code>popup_list</code>	NULL
<code>num_popups</code>	0
<code>name</code>	name argument to <code>XtCreateWidget</code> (may not be changed)
<code>screen</code>	Parent's screen, top-level widget gets it from display specifier (may not be changed)
<code>colormap</code>	Default color map for the screen
<code>window</code>	NULL
<code>depth</code>	Parent's depth, top-level widget gets root window depth
<code>background_pixel</code>	XtDefaultBackground
<code>background_pixmap</code>	NULL
<code>visible</code>	True
<code>map_when_managed</code>	True

1.3.2 Composite Widgets

Composite widgets are a subclass of the `Core` widget (see Chapter 3) are intended to be containers for other widgets, and are defined by the `CompositeClassPart` and `CompositePart` structures.

1.3.2.1 CompositeClassPart Structure

– In addition to the `Core` widget class fields, `Composite` widgets have the following class fields:

```
typedef struct {
    XtGeometryHandler geometry_manager;    See Chapter 6
    XtWidgetProc change_managed;         See Chapter 3
    XtWidgetProc insert_child;          See Chapter 3
    XtWidgetProc delete_child;          See Chapter 3
    caddr_t extension;                   See Section 1.4
} CompositeClassPart;
```


Composite widget classes have the composite fields immediately following the core fields:

```
typedef struct {
    CoreClassPart          core_class;
    CompositeClassPart     composite_class;
} CompositeClassRec, *CompositeWidgetClass;
```

The predefined class record and pointer for `CompositeClassRec` are:

```
extern CompositeClassRec compositeClassRec;
extern WidgetClass compositeWidgetClass;
```

The opaque types `CompositeWidget` and `CompositeWidgetClass` and the opaque variable `compositeWidgetClass` are defined for generic operations on widgets that are a subclass of `CompositeWidget`.

1.3.2.2 CompositePart Structure

– In addition to the `CorePart` fields, `Composite` widgets have the following fields defined in the `CompositePart` structure:

```
typedef struct {
    WidgetList children;           See Section 1.4
    Cardinal num_children;        See Section 1.4
    Cardinal num_slots;           See Chapter 3
    XtOrderProc insert_position;  See Section 2.4
} CompositePart;
```

Composite widgets have the composite fields immediately following the core fields:

```
typedef struct {
    CorePart core;
    CompositePart composite;
} CompositeRec, *CompositeWidget;
```

1.3.2.3 CompositePart Default Values

– The default values for the composite fields, which are filled in by the `Composite` resource list and the `Composite` initialize procedure, are:

Field	Default Value
children	NULL
num_children	0
num_slots	0

Field	Default Value
insert_position	Internal function InsertAtEnd

1.3.3 Constraint Widgets

Constraint widgets are a subclass of the Composite widget (see Section 3.7) that maintain additional state data for each child, for example, client-defined constraints on the child's geometry. They are defined by the `ConstraintClassPart` and `ConstraintPart` structures.

1.3.3.1 ConstraintClassPart Structure

– In addition to the Composite class fields, Constraint widgets have the following class fields:

```
typedef struct {
    XtResourceList resources;           See Section 3.7
    Cardinal num_resources;           See Section 3.7
    Cardinal constraint_size;         See Section 3.7
    XtInitProc initialize;           See Section 3.7
    XtWidgetProc destroy;           See Section 3.7
    XtSetValuesFunc set_values;       See Section 3.7
    caddr_t extension;               See Section 1.4
} ConstraintClassPart;
```

Constraint widget classes have the constraint fields immediately following the composite fields:

```
typedef struct {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ConstraintClassPart constraint_class;
} ConstraintClassRec, *ConstraintWidgetClass;
```

The predefined class record and pointer for `ConstraintClassRec` are:

```
extern ConstraintClassRec constraintClassRec;
extern WidgetClass constraintWidgetClass;
```

The opaque types `ConstraintWidget` and `ConstraintWidgetClass` and the opaque variable `constraintWidgetClass` are defined for generic operations on widgets that are a subclass of `ConstraintWidgetClass`.

1.3.3.2 ConstraintPart Structure

– In addition to the `CompositePart` fields, `Constraint` widgets have the following fields defined in the `ConstraintPart` structure:

```
typedef struct { int empty; } ConstraintPart;
```

`Constraint` widgets have the constraint fields immediately following the composite fields:

```
typedef struct {  
    CorePart core;  
    CompositePart composite;  
    ConstraintPart constraint;  
} ConstraintRec, *ConstraintWidget;
```

1.4 Widget Classing

The `widget_class` field of a widget points to its widget class structure, which contains information that is constant across all widgets of that class. As a consequence, widget classes usually do not implement directly callable procedures; rather, they implement procedures that are available through their widget class structure. These methods are invoked by generic procedures that envelop common actions around the procedures implemented by the widget class. Such procedures are applicable to all widgets of that class and also to widgets that are subclasses of that class.

All widget classes are a subclass of `Core` and can be subclassed further. Subclassing reduces the amount of code and declarations you write to make a new widget class that is similar to an existing class. For example, you do not have to describe every resource your widget uses in an `XtResourceList`. Instead, you describe only the resources your widget has that its superclass does not. Subclasses usually inherit many of their superclass's procedures (for example, the `expose` procedure or `geometry` handler).

Subclassing, however, can be taken too far. If you create a subclass that inherits none of the procedures of its superclass, you should consider whether or not you have chosen the most appropriate superclass.

To make good use of subclassing, widget declarations and naming conventions are highly stylized. A widget consists of three files:

- A public `.h` file that is used by client widgets or applications
- A private `.h` file that is used by widgets that are subclasses of the widget
- A `.c` file that implements the widget class

1.4.1 Widget Naming Conventions

The Intrinsics provide a vehicle by which programmers can create new widgets and organize a collection of widgets into an application. To ensure that applications need not deal with as many styles of capitalization and spelling as the number of widget classes it uses, the following guidelines should be followed when writing new widgets:

- Use the X naming conventions that are applicable. For example, a record component name is all lowercase and uses underscores (_) for compound words (for example, `background_pixmap`). Type and procedure names start with uppercase and use capitalization for compound words (for example, `ArgList` or `XtSetValues`).
- A resource name string is spelled identically to the field name except that compound names use capitalization rather than underscore. To let the compiler catch spelling errors, each resource name should have a macro definition prefixed with `XtN`. For example, the `background_pixmap` field has the corresponding resource name identifier `XtNbackgroundPixmap`, which is defined as the string “backgroundPixmap”. Many predefined names are listed in `<X11/StringDefs.h>`. Before you invent a new name, you should make sure that your proposed name is not already defined or that there is not already a name that you can use.
- A resource class string starts with a capital letter and uses capitalization for compound names (for example, “BorderWidth”). Each resource class string should have a macro definition prefixed with `XtC` (for example, `XtCBorderWidth`).
- A resource representation string is spelled identically to the type name (for example, “TranslationTable”). Each representation string should have a macro definition prefixed with `XtR` (for example, `XtRTranslationTable`).
- New widget classes start with a capital and use uppercase for compound words. Given a new class name `AbcXyz` you should derive several names:
 - Partial widget instance structure name `AbcXyzPart`
 - Complete widget instance structure names `AbcXyzRec` and `_AbcXyzRec`
 - Widget instance pointer type name `AbcXyzWidget`
 - Partial class structure name `AbcXyzClassPart`
 - Complete class structure names `AbcXyzClassRec` and `_AbcXyzClassRec`
 - Class structure variable `abcXyzClassRec`
 - Class pointer variable `abcXyzWidgetClass`

- Action procedures available to translation specifications should follow the same naming conventions as procedures. That is, they start with a capital letter and compound names use uppercase (for example, “Highlight” and “NotifyClient”).

1.4.2 Widget Subclassing in Public .h Files

The public .h file for a widget class is imported by clients and contains:

- A reference to the public .h files for the superclass
- The names and classes of the new resources that this widget adds to its superclass
- The class record pointer that you use to create widget instances
- The C type that you use to declare widget instances of this class
- Entry points for new class methods

For example, the following is the public .h file for a possible implementation of a Label widget:

```
#ifndef LABEL_H
#define LABEL_H

/* New resources */
#define XtNjustify          "justify"
#define XtNforeground      "foreground"
#define XtNlabel           "label"
#define XtNfont            "font"
#define XtNinternalWidth  "internalWidth"
#define XtNinternalHeight "internalHeight"

/* Class record pointer */
extern WidgetClass labelWidgetClass;

/* C Widget type definition */
typedef struct _LabelRec    *LabelWidget;

/* New class method entry points */
extern void Label SetText();
    /* Widget w */
    /* String text */

extern String Label GetText();
    /* Widget w */

#endif LABEL_H
```

The conditional inclusion of the text allows the application to include header files for different widgets without being concerned that they already may be included as a superclass of another widget.

To accommodate operating systems with file name length restrictions, the name of the public .h file is the first ten characters of the widget class. For example, the public .h file for the Constraint widget is Constraint.h.

1.4.3 Widget Subclassing in Private .h Files

The private .h file for a widget is imported by widget classes that are subclasses of the widget and contains:

- A reference to the public .h file for the class
- A reference to the private .h file for the superclass
- The new fields that the widget instance adds to its superclass's widget structure
- The complete widget instance structure for this widget
- The new fields that this widget class adds to its superclass's Constraint structure if the widget is a subclass of Constraint
- The complete Constraint structure if the widget is a subclass of Constraint
- The new fields that this widget class adds to its superclass's widget class structure
- The complete widget class structure for this widget
- The name of a constant of the generic widget class structure
- An inherit procedure for subclasses that wish to inherit a superclass operation for each new procedure in the widget class structure

For example, the following is the private .h file for a possible Label widget:

```
#ifndef LABELP_H
#define LABELP_H

#include <X11/Label.h>

/* New fields for the Label widget record */
typedef struct {
/* Settable resources */
    Pixel foreground;
    XFontStruct *font;
    String label;
    XtJustify justify;
    /* text to display */
};
```

```

        Dimension internal_width;           /* # of pixels horizontal border */
        Dimension internal_height;         /* # of pixels vertical border */

/* Data derived from resources */
    GC normal_GC;
    GC gray_GC;
    Pixmap gray_pixmap;
    Position label_x;
    Position label_y;
    Dimension label_width;
    Dimension label_height;
    Cardinal label_len;
    Boolean display_sensitive;
} LabelPart;

/* Full instance record declaration */
typedef struct _LabelRec {
    CorePart core;
    LabelPart label;
} LabelRec;

/* Types for label class methods */
typedef void (*LabelSetTextProc)();
    /* Widget w */
    /* String text */

typedef String (*LabelGetTextProc)();
    /* Widget w */

/* New fields for the Label widget class record */
typedef struct {
    LabelSetTextProc set_text;
    LabelGetTextProc get_text;
    caddr_t extension;
} LabelClassPart;

/* Full class record declaration */
typedef struct _LabelClassRec {
    CoreClassPart core_class;
    LabelClassPart label_class;
} LabelClassRec;

/* Class record variable */
extern LabelClassRec labelClassRec;

```

```
#define LabelInheritSetText((LabelSetTextProc)_XtInherit)
#define LabelInheritGetText((LabelGetTextProc)_XtInherit)
#endif LABELP_H
```

To accommodate operating systems with file name length restrictions, the name of the private .h file is the first nine characters of the widget class followed by a capital P. For example, the private .h file for the Constraint widget is `ConstrainP.h`.

1.4.4 Widget Subclassing in .c Files

The .c file for a widget contains the structure initializer for the class record variable, which contains the following parts:

- Class information (for example, superclass, class_name, widget_size, class_initialize, and class_ited)
- Data constants (for example, resources and num_resources, actions and num_actions, visible_interest, compress_motion, compress_exposure, and version)
- Widget operations (for example, initialize, realize, destroy, resize, expose, set_values, accept_focus, and any operations specific to the widget)

The superclass field points to the superclass `WidgetClass` record. For direct subclasses of the generic core widget, superclass should be initialized to the address of the `widgetClassRec` structure. The superclass is used for class chaining operations and for inheriting or enveloping a superclass's operations. (See Sections 1.4.7, 1.4.9, and 1.4.10).

The class_name field contains the text name for this class (used by the resource manager). For example, the Label widget has the string "Label". More than one widget class can share the same text class name.

The widget_size field is the size of the corresponding widget structure (not the size of the Class structure).

The version field indicates the toolkit version number and is used for run-time consistency checking of the XUI Toolkit and widgets in an application. Widget writers must set it to the symbolic value `XtVersion` in the widget class initialization. Those widget writers who know that their widgets are backwards compatible with previous versions of the Intrinsics can put the special value `XtVersionDontCheck` in the version field to turn off version checking for those widgets.

The extension field is for future upwards compatibility. If you add additional fields to class parts, all subclass structure layouts change, requiring complete recompilation. To allow clients to avoid recompilation, an extension field at the end of each class part can point to a record that

contains any additional class information required.

All other fields are described in their respective sections.

The following is an abbreviated version of the .c file for the Label widget. (The resources table is described in the Chapter 9.)

```
/* Resources specific to Label */
#define XtRJustify      "Justify"
static XtResource resources[] = {
    {XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
     XtOffset(LabelWidget, label.foreground), XtRString, XtDefaultForeground},
    {XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct *),
     XtOffset(LabelWidget, label.font), XtRString, XtDefaultFont},
    {XtNlabel, XtCLabel, XtRString, sizeof(String),
     XtOffset(LabelWidget, label.label), XtRString, NULL},
    .
    .
    .
}

/* Forward declarations of procedures */
static void ClassInitialize();
static void Initialize();
static void Realize();
static void SetText();
static void GetText();
.
.
.

/* Class record constant */
LabelClassRec labelClassRec = {
    {
        /* core_class fields */
        /* superclass          */ (WidgetClass) &widgetClassRec,
        /* class_name          */ "Label",
        /* widget_size         */ sizeof(LabelRec),
        /* class_initialize    */ ClassInitialize,
        /* class_part_initialize */ NULL,
        /* class_inited        */ False,
        /* initialize          */ Initialize,
        /* initialize_hook     */ NULL,
        /* realize              */ Realize,
        /* actions              */ NULL,
        /* num_actions         */ 0,
        /* resources           */ resources,
    }
}
```

```

        /* num_resources          */          XtNumber(resources),
        /* xrm_class              */          NULLQUARK,
        /* compress_motion        */          True,
        /* compress_exposure      */          True,
        /* compress_enterleave    */          True,
        /* visible_interest       */          False,
        /* destroy                 */          NULL,
        /* resize                  */          Resize,
        /* expose                  */          Redisplay,
        /* set_values              */          SetValues,
        /* set_values_hook         */          NULL,
        /* set_values_almost      */          XtInheritSetValuesAlmost,
        /* get_values_hook         */          NULL,
        /* accept_focus           */          NULL,
        /* version                 */          XtVersion,
        /* callback_offsets        */          NULL,
        /* tm_table                */          NULL,
        /* query_geometry          */          XtInheritQueryGeometry,
        /* display_accelerator     */          NULL,
        /* extension               */          NULL
    },
    {
        /* Label_class fields      */
        /* get_text                */          GetText,
        /* set_text                */          SetText,
        /* extension               */          NULL
    }
};

```

```

/* Class record pointer */

```

```

WidgetClass labelWidgetClass = (WidgetClass) &labelClassRec;

```

```

/* New method access routines */

```

```

void Label SetText(w, text)

```

```

    Widget w;
    String text;

```

```

{

```

```

    Label WidgetClass lwc = (Label WidgetClass)XtClass(w);
    XtCheckSubclass(w, labelWidgetClass, NULL);
    *(lwc->label_class.set_text)(w, text)

```

```

}

```

```

/* Private procedures */

```

```

.
.
.

```

1.4.5 Widget Class and Superclass Lookup

To obtain the class of a widget, use `XtClass`.

```
WidgetClass XtClass(w)
    Widget w;
```

w Specifies the widget.

The `XtClass` function returns a pointer to the widget's class structure.

To obtain the superclass of a widget, use `XtSuperclass`.

```
WidgetClass XtSuperclass(w)
    Widget w;
```

w Specifies the widget.

The `XtSuperclass` function returns a pointer to the widget's superclass class structure.

1.4.6 Widget Subclass Verification

To check the subclass that a widget belongs to, use `XtIsSubclass`.

```
Boolean XtIsSubclass(w, widget_class)
    Widget w;
    WidgetClass widget_class;
```

w Specifies the widget.

widget_class Specifies the widget class to test against.

The `XtIsSubclass` function returns `True` if the class of the specified widget is equal to or is a subclass of the specified widget class. The specified widget can be any number of subclasses down the chain and need not be an immediate subclass of the specified widget class. Composite widgets that need to restrict the class of the items they contain can use `XtIsSubclass` to find out if a widget belongs to the desired class of objects.

To check the subclass that a widget belongs to and generate a debugging error message, use `XtCheckSubclass`.

```
void XtCheckSubclass(w, widget_class, message)
    Widget w;
    WidgetClass widget_class;
    String message;
```

w Specifies the widget.

widget_class Specifies the widget class to test against.

message Specifies the message that is to be used.

The `XtCheckSubclass` macro determines if the class of the specified widget is equal to or is a subclass of the specified widget class. The widget can be any number of subclasses down the chain and need not be an immediate subclass of the specified widget class. If the specified widget is not a subclass, `XtCheckSubclass` constructs an error message from the supplied message, the widget's actual class, and the expected class and calls `XtErrorMsg`. `XtCheckSubclass` should be used at the entry point of exported routines to ensure that the client has passed in a valid widget class for the exported operation.

`XtCheckSubclass` is only executed when the widget has been compiled with the compiler symbol `DEBUG` defined; otherwise, it is defined as the empty string and generates no code.

1.4.7 Superclass Chaining

While most fields in a widget class structure are self-contained, some fields are linked to their corresponding field in their superclass or subclass structures. With a linked field, the Intrinsics access its value only after accessing its corresponding superclass value (called downward superclass chaining) or before accessing its corresponding superclass value (called upward superclass chaining). The self-contained fields in a widget class are:

- `class_name`
- `class_initialize`
- `widget_size`
- `realize`
- `visible_interest`
- `resize`
- `expose`
- `accept_focus`
- `compress_motion`

- `compress_exposure`
- `compress_enterleave`
- `set_values_almost`
- `tm_table`
- `version`

With downward superclass chaining, the invocation of an operation first accesses the field from the `Core` class structure, then the subclass structure, and so on down the class chain to that widget's class structure. These superclass-to-subclass fields are:

- `class_part_initialize`
- `get_values_hook`
- `initialize`
- `initialize_hook`
- `set_values`
- `set_values_hook`
- `resources`

In addition, for subclasses of `Constraint`, the `resources` field of the `ConstraintClassPart` structure is chained from the `Constraint` class down to the subclass.

With upward superclass chaining, the invocation of an operation first accesses the field from the widget class structure, then the field from the superclass structure, and so on up the class chain to the `Core` class structure. The subclass-to-superclass fields are:

- `destroy`
- `actions`

1.4.8 Class Initialization: `class_initialize` and `class_part_initialize` Procedures

Many class records can be initialized completely at compile time. In some cases, however, a class may need to register type converters or perform other sorts of one-time initialization.

Because the C language does not have initialization procedures that are invoked automatically when a program starts up, a widget class can declare a `class_initialize` procedure that will be automatically called exactly once by the XUI Toolkit. A class initialization procedure pointer is of type `XtProc`:

```
typedef void (*XtProc)();
```

A widget class indicates that it has no class initialization procedure by specifying NULL in the `class_initialize` field.

In addition to having class initializations done exactly once, some classes need to perform additional initialization for fields in its part of the class record. These are performed not just for the particular class but for subclasses as well. This is done in the class's class part initialization procedure, which is stored in the `class_part_initialize` field. The `class_part_initialize` procedure pointer is of type `XtWidgetClassProc`:

```
typedef void (*XtWidgetClassProc)(WidgetClass);
```

During class initialization, the class part initialization procedure for the class and all its superclasses are called in superclass-to-subclass order on the class record. These procedures have the responsibility of doing any dynamic initializations necessary to their class's part of the record. The most common is the resolution of any inherited methods defined in the class. For example, if a widget class C has superclasses `Core`, `Composite`, `A`, and `B`, the class record for C first is passed to `Core`'s `class_part_initialize` record. This resolves any inherited core methods and compiles the textual representations of the resource list and action table that are defined in the class record. Next, the `Composite`'s `class_part_initialize` is called to initialize the composite part of C's class record. Finally, the `class_part_initialize` procedures for `A`, `B`, and `C` (in order) are called. For further information, see Section 1.4.9. Classes that do not define any new class fields or that need no extra processing for them can specify NULL in the `class_part_initialize` field.

All widget classes, whether they have a class initialization procedure or not, must start with their `class_inited` field `False`.

The first time a widget of a class is created, `XtCreateWidget` ensures that the widget class and all superclasses are initialized, in superclass to subclass order, by checking each `class_inited` field and if it is `False`, by calling the `class_initialize` and the `class_part_initialize` procedures for the class and all its superclasses. The `Intrinsics` then set the `class_inited` field to `True`. After the one-time initialization, a class structure is constant.

The following provides the class initialization procedure for `Label`.

```
static void ClassInitialize()
{
    XtQEleft   = XrmStringToQuark("left");
    XtQEcenter = XrmStringToQuark("center");
    XtQErighth = XrmStringToQuark("right");

    XtAddConverter(XtRString, XtRJustify, CvtStringToJustify, NULL, 0);
}
```

A class is initialized the first time a widget of that class or any subclass is created. If the class initialization procedure registers type converters, these type converters are not available until this first widget is created (see Section 9.6).

1.4.9 Inheritance of Superclass Operations

A widget class is free to use any of its superclass's self-contained operations rather than implementing its own code. The most frequently inherited operations are:

- `expose`
- `realize`
- `insert_child`
- `delete_child`
- `geometry_manager`
- `set_values_almost`

To inherit an operation `xyz`, specify the constant `XtInheritXyz` in your class record.

Every class that declares a new procedure in its widget class part must provide for inheriting the procedure in its `class_part_initialize` procedure. (The special chained operations `initialize`, `set_values`, and `destroy` declared in the Core record do not have inherit procedures. Widget classes that do nothing beyond what their superclass does specify `NULL` for chained procedures in their class records.)

Inheriting works by comparing the value of the field with a known, special value and by copying in the superclass's value for that field if a match occurs. This special value is usually the Intrinsic internal value `_XtInherit` cast to the appropriate type. (`_XtInherit` is a procedure that issues an error message if it is actually called.)

For example, the Composite class's private include file contains these definitions:

```
#define XtInheritGeometryManager ((XtGeometryHandler) _XtInherit)
#define XtInheritChangeManaged ((XtWidgetProc) _XtInherit)
#define XtInheritInsertChild ((XtArgsProc) _XtInherit)
#define XtInheritDeleteChild ((XtWidgetProc) _XtInherit)
```

The Composite's `class_part_initialize` procedure begins as follows:

```
static void CompositeClassPartInitialize(widgetClass)
    WidgetClass widgetClass;
{
    register CompositeWidgetClass wc = (CompositeWidgetClass) widgetClass;
```

```

CompositeWidgetClass super = (CompositeWidgetClass) wc->core_class.superclass

if (wc->composite_class.geometry_manager == XtInheritGeometryManager) {
    wc->composite_class.geometry_manager = super->composite_class.geometry_manager;
}

if (wc->composite_class.change_managed == XtInheritChangeManaged) {
    wc->composite_class.change_managed = super->composite_class.change_managed;
}
.
.
.

```

The inherit constants defined for Core are:

- XtInheritRealize
- XtInheritResize
- XtInheritExpose
- XtInheritSetValuesAlmost
- XtInheritAcceptFocus
- XtInheritDisplayAccelerator

The inherit constants defined for Composite are:

- XtInheritGeometryManager
- XtInheritChangeManaged
- XtInheritInsertChild
- XtInheritDeleteChild

1.4.10 Invocation of Superclass Operations

A widget class sometimes explicitly needs to call a superclass operation that usually is not chained. For example, a widget's expose procedure might call its superclass's expose and then perform a little more work of its own. Composite classes with fixed children can implement insert_child by first calling their superclass's insert_child procedure and then calling XtManageChild to add the child to the managed list.

Note that a method should call its own superclass method, not the widget's superclass method. That is, it should use its own class pointers only, not the widget's class pointers. This technique is referred to as *enveloping* the superclass's operation.

Widget Instantiation 2

A collection of widget instances constitutes a widget tree. The shell widget returned by `XtAppCreateShell` is the root of the widget tree instance. The widgets with one or more children are the intermediate nodes of that tree, and the widgets with no children of any kind are the leaves of a widget tree. With the exception of pop-up children (see Chapter 5), this widget tree instance defines the associated X Window tree.

Widgets can be either composite or primitive. Both kinds of widgets can contain children, but the `Intrinsics` provide a set of management mechanisms for constructing and interfacing between composite widgets, their children, and other clients.

Composite widgets, subclasses of `Composite`, are containers for an arbitrary but implementation-defined collection of children, which may be instantiated by the composite widget itself, by other clients, or by a combination of the two. Composite widgets also contain methods for managing the geometry (layout) of any child widget. Under unusual circumstances, a composite widget may have zero children, but it usually has at least one. By contrast, primitive widgets that contain children typically instantiate specific children of known class themselves and do not expect external clients to do so. Primitive widgets also do not have general geometry management methods.

In addition, the `Intrinsics` recursively perform many operations (for example, realization and destruction) on composite widgets and all of their children. Primitive widgets that have children must be prepared to perform the recursive operations themselves on behalf of their children.

A widget tree is manipulated by several `Intrinsics` functions. For example, `XtRealizeWidget` traverses the tree downward and recursively realizes all pop-up widgets and children of composite widgets. `XtDestroyWidget` traverses the tree downward and destroys all pop-up widgets and children of composite widgets. The functions that fetch and modify resources traverse the tree upward and determine the inheritance of resources from a widget's ancestors. `XtMakeGeometryRequest` traverses the tree up one level and calls the geometry manager that is responsible for a widget child's geometry.

To facilitate up-traversal of the widget tree, each widget has a pointer to its parent widget. The Shell widget that `XtAppCreateShell` returns, however, has a parent pointer of `NULL`.

To facilitate down-traversal of the widget tree, each composite widget has a pointer to an array of children widgets, which includes all normal children created, not just the subset of children that are managed by the composite widget's geometry manager. Primitive widgets that instantiate children are entirely responsible for all operations that require downward traversal below themselves. In addition, every widget has a pointer to an array of pop-up children widgets.

2.1 Initializing the XUI Toolkit

Before an application can call any of the Intrinsic functions, it must initialize the XUI Toolkit by using:

- `XtToolkitInitialize`, which initializes the XUI Toolkit internals
- `XtCreateApplicationContext`, which initializes the per application state
- `XtDisplayInitialize` or `XtOpenDisplay`, which initializes the per display state
- `XtAppCreateShell`, which creates the initial widget

Multiple instances of XUI Toolkit applications may be implemented by a single program in a single address space. Each instance needs to be able to read input and dispatch events independently of any other instance. Further, an application may need multiple display connections or need to have widgets on multiple screens. To accommodate both requirements, the Intrinsic functions define application contexts, each of which provides the information needed to distinguish one application instance from another. The major component of an application context is a list of X Display pointers for that application. The application context type `XtAppContext` is opaque to clients.

To initialize the XUI Toolkit internals, use `XtToolkitInitialize`.

```
void XtToolkitInitialize()
```

The semantics of calling `XtToolkitInitialize` more than once are undefined.

To create an application context, use `XtCreateApplicationContext`.

```
XtAppContext XtCreateApplicationContext()
```

The `XtCreateApplicationContext` function returns an application context, which is an opaque type. Every application must have at least one application context.

To destroy an application context and close any displays in it, use `XtDestroyApplicationContext`.

```
void XtDestroyApplicationContext(app_context)
    XtAppContext app_context;
```

app_context Specifies the application context.

The `XtDestroyApplicationContext` function destroys the specified application context as soon as it is safe to do so. If called from within an event dispatch (for example, a callback procedure), `XtDestroyApplicationContext` does not destroy the application context until the dispatch is complete.

To get the application context for a given widget, use `XtWidgetToApplicationContext`.

```
XtAppContext XtWidgetToApplicationContext(w)
    Widget w;
```

w Specifies the widget for which you want the application context .

The `XtWidgetToApplicationContext` function returns the application context for the specified widget.

To initialize a display and add it to an application context, use `XtDisplayInitialize`.

```
void XtDisplayInitialize(app_context, display, application_name,
                        application_class, options, num_options,
                        argc, argv)
    XtAppContext app_context;
    Display *display;
    String application_name;
    String application_class;
    XrmOptionDescRec *options;
    Cardinal num_options;
    Cardinal *argc;
    String *argv;
```

- app_context* Specifies the application context.
- display* Specifies the display. Note that a display can be in at most one application context.
- application_name*
Specifies the name of the application instance.
- application_class*
Specifies the class name of this application, which is usually the generic name for all instances of this application.
- options* Specifies how to parse the command line for any application-specific resources. The options argument is passed as a parameter to XrmParseCommand. For further information, see the *Guide to the Xlib Library*.
- num_options* Specifies the number of entries in the options list.
- argc* Specifies a pointer to the number of command line parameters.
- argv* Specifies the command line parameters.

The XtDisplayInitialize function builds the resource database, calls the Xlib XrmParseCommand function to parse the command line, and performs other per display initialization. After XrmParseCommand has been called, argc and argv contain only those parameters that were not in the standard option table or in the table specified by the options argument. If the modified argc is not zero, most applications simply print out the modified argv along with a message listing the allowable options. On UNIX-based systems, the application name is usually the final component of argv[0]. If the synchronize resource is True for the specified application, XtDisplayInitialize calls the Xlib XSynchronize function to put Xlib into synchronous mode for this display connection. If the reverseVideo resource is True, the Intrinsics exchange XtDefaultForeground and XtDefaultBackground for widgets created on this display. (See Section 9.6.1).

To open a display, initialize it, and add it to an application context, use XtOpenDisplay.

```
Display *XtOpenDisplay(app_context, display_string,
                     application_name, application_class,
                     options, num_options, argc, argv)
XtAppContext app_context;
String display_string;
String application_name;
```

(continued on next page)

```
String application_class;  
XrmOptionDescRec *options;  
Cardinal num_options;  
Cardinal *argc;  
String *argv;
```

app_context Specifies the application context.

display_string Specifies the display string. Note that a display can be in at most one application context.

application_name
Specifies the name of the application instance.

application_class
Specifies the class name of this application, which is usually the generic name for all instances of this application.

options Specifies how to parse the command line for any application-specific resources. The options argument is passed as a parameter to `XrmParseCommand`. For further information, see the *Guide to the Xlib Library*.

num_options Specifies the number of entries in the options list.

argc Specifies a pointer to the number of command line parameters.

argv Specifies the command line parameters.

The `XtOpenDisplay` function calls `XOpenDisplay` the specified display name. If `display_string` is `NULL`, `XtOpenDisplay` uses the current value of the `-display` option specified in `argv` and if no display is specified in `argv`, uses the user's default display (on UNIX-based systems, this is the value of the `DISPLAY` environment variable).

If this succeeds, it then calls `XtDisplayInitialize` and pass it the opened display and the value of the `-name` option specified in `argv` as the application name. If no name option is specified, it uses the application name passed to `XtOpenDisplay`. If the application name is `NULL`, it uses the last component of `argv[0]`. `XtOpenDisplay` returns the newly opened display or `NULL` if it failed.

`XtOpenDisplay` is provided as a convenience to the application programmer.

To close a display and remove it from an application context, use `XtCloseDisplay`.

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

display Specifies the display.

The XtCloseDisplay function closes the specified display as soon as it is safe to do so. If called from within an event dispatch (for example, a callback procedure), XtCloseDisplay does not close the display until the dispatch is complete. Note that applications need only call XtCloseDisplay if they are to continue executing after closing the display; otherwise, they should call XtDestroyApplicationContext or just exit.

2.2 Loading the Resource Database

The XtDisplayInitialize function loads the application's resource database for this display/host/application combination from the following sources (in order):

- Application-specific class resource file on the local host
- Application-specific user resource file on the local host
- Resource property on the server or user preference resource file on the local host
- Per-host user environment resource file on the local host
- Application command line (*argv*)

Each resource database is kept on a per-display basis.

The application-specific class resource file name is constructed from the class name of the application. It points to a site-specific resource file that usually is installed by the site manager when the application is installed. On UNIX-based systems, this file usually is `/usr/lib/X11/app-defaults/class`, where *class* is the application class name. This file is expected to be provided by the developer of the application and may be required for the application to function properly.

The application-specific user resource file name is constructed from the class name of the application and points to a user-specific resource file. This file is owned by the application and typically stores user customizations. On UNIX-based systems, this file name is constructed from the user's XAPPLRESDIR variable by appending *class* to it, where *class* is the application class name. If XAPPLRESDIR is not defined, it defaults to the user's home directory. If the resulting resource file exists, it is merged into the resource database. This file may be provided with the application or constructed by the user.

The server resource file is the contents of the X server's RESOURCE_MANAGER property that was returned by XOpenDisplay. If no such property exists for the display, the contents of the resource file in

the user's home directory is used instead. On UNIX-based systems, the usual name for the user preference resource file is `.Xdefaults`. If the resulting resource file exists, it is merged into the resource database. The server resource file is constructed entirely by the user and contains both display-independent and display-specific user preferences.

If one exists, a user's environment resource file is then loaded and merged into the resource database. This file name is user and host specific. On UNIX-based systems, the user's environment resource file name is constructed from the value of the user's `XENVIRONMENT` variable for the full path of the file. If this environment variable does not exist, `XtDisplayInitialize` searches the user's home directory for the `.Xdefaults-host` file, where *host* is the name of the machine on which the application is running. If the resulting resource file exists, it is merged into the resource database. The environment resource file is expected to contain process-specific resource specifications that are to supplement those user-preference specifications in the server resource file.

To obtain the resource database for a particular display, use `XtDatabase`.

```
XrmDatabase XtDatabase(display)
    Display *display;
```

display Specifies the display.

The `XtDatabase` function returns the fully merged resource database that was built by `XtDisplayInitialize` associated with the display that was passed in. If this display has not been initialized by `XtDisplayInitialize`, the results are not defined.

2.3 Parsing the Command Line

The `XtOpenDisplay` function first parses the command line for the following options:

- `-display` Specifies the display name for `XOpenDisplay`, which overrides the display name passed to `XtDisplayInitialize`.
- `-name` Sets the resource name prefix, which overrides the application name passed to `XtDisplayInitialize`.

`XtDisplayInitialize` has a table of standard command line options that are passed to `XrmParseCommand` for adding resources to the resource database, and it takes as a parameter additional application-specific resource abbreviations. The format of this table is:

```
typedef enum {
    XrmoptionNoArg,          /* Value is specified in OptionDescRec.value */
    XrmoptionIsArg,         /* Value is the option string itself */
```



```

        XrmoptionStickyArg,      /* Value is characters immediately following option */
        XrmoptionSepArg,        /* Value is next argument in argv */
        XrmoptionSkipArg,      /* Ignore this option and the next argument in argv */
        XrmoptionSkipLine      /* Ignore this option and the rest of argv */
} XrmOptionKind;

typedef struct {
    char *option;                /* Option name in argv */
    char *specifier;            /* Resource name (without application name) */
    XrmOptionKind argKind;      /* Which style of option it is */
    caddr_t value;              /* Value to provide if XrmoptionNoArg */
} XrmOptionDescRec, *XrmOptionDescList;

```

The standard table contains the following entries:

Option String	Resource Name	Argument Kind	Resource Value
- background	background	SepArg	next argument
- bd	borderColor	SepArg	next argument
- bg	background	SepArg	next argument
- borderwidth	borderWidth	SepArg	next argument
- bordercolor	borderColor	SepArg	next argument
- bw	borderWidth	SepArg	next argument
- display	display	SepArg	next argument
- fg	foreground	SepArg	next argument
- fn	font	SepArg	next argument
- font	font	SepArg	next argument
- foreground	foreground	SepArg	next argument
- geometry	geometry	SepArg	next argument
- iconic	iconic	NoArg	true
- name	name	SepArg	next argument
- reverse	reverseVideo	NoArg	on
- rv	reverseVideo	NoArg	on
+ rv	reverseVideo	NoArg	off
- selectionTimeout	selectionTimeout	SepArg	next argument
- synchronous	synchronize	NoArg	on
+ synchronous	synchronize	NoArg	off
- title	title	SepArg	next argument
- xrm	next argument	ResArg	next argument

Note that any unique abbreviation for an option name in the standard table or in the application table is accepted.

If `reverseVideo` is set, the values of `XtDefaultForeground` and `XtDefaultBackground` are exchanged. If `synchronize` is set, the `Intrinsics` put `Xlib` into synchronous mode for all connections.

The `-xrm` option provides a method of setting any resource in an application. The next argument should be a quoted string identical in format to a line in the user resources file. For example, to give a red background to all command buttons in an application named `xmh`, you can start it up as:

```
xmh -xrm 'xmh*Command.background: red'
```

When it parses the command line, `XtDisplayInitialize` merges the application option table with the standard option table before calling the `Xlib XrmParseCommand` function. An entry in the application table with the same name as an entry in the standard table overrides the standard table entry. If an option name is a prefix of another option name, both names are kept in the merged table.

2.4 Creating Widgets

The creation of widget instances is a three-phase process:

1. The widgets are allocated and initialized with resources and are optionally added to the managed subset of their parent.
2. All composite widgets are notified of their managed children in a bottom-up traversal of the widget tree.
3. The widgets create X windows that then get mapped.

To start the first phase, the application calls `XtCreateWidget` for all its widgets and adds some (usually, most or all) of its widgets to their respective parent's managed set by calling `XtManageChild`. To avoid an $O(n^2)$ creation process where each composite widget lays itself out each time a widget is created and managed, parent widgets are not notified of changes in their managed set during this phase.

After all widgets have been created, the application calls `XtRealizeWidget` on the top-level widget to start the second and third phases. `XtRealizeWidget` first recursively traverses the widget tree in a post-order (bottom-up) traversal and then notifies each composite widget with one or more managed children by means of its `change_managed` procedure.

Notifying a parent about its managed set involves geometry layout and possibly geometry negotiation. A parent deals with constraints on its size imposed from above (for example, when a user specifies the application window size) and suggestions made from below (for example, when a primitive child computes its preferred size). One difference between the two can cause geometry changes to ripple in both directions through the

widget tree. The parent may force some of its children to change size and position and may issue geometry requests to its own parent in order to better accommodate all its children. You cannot predict where anything will go on the screen until this process finishes.

Consequently, in the first and second phases, no X windows are actually created because it is likely that they will get moved around after creation. This avoids unnecessary requests to the X server.

Finally, `XtRealizeWidget` starts the third phase by making a pre-order (top-down) traversal of the widget tree, allocates an X window to each widget by means of its realize procedure, and finally maps the widgets that are managed.

2.4.1 Creating and Merging Argument Lists

Many Intrinsics functions need to be passed pairs of resource names and values. These are passed as an `ArgList`, which contains:

```
typedef something XtArgVal;

typedef struct {
    String name;
    XtArgVal value;
} Arg, *ArgList;
```

Where *something* is a type large enough to contain `caddr_t`, `char *`, `long`, `int *`, or a pointer to a function.

If the size of the resource is less than or equal to the size of an `XtArgVal`, the resource value is stored directly in `value`; otherwise, a pointer to it is stored into `value`.

To set values in an `ArgList`, use `XtSetArg`.

```
XtSetArg(arg, name, value)
    Arg arg;
    String name;
    XtArgVal value;
```

arg Specifies the name-value pair to set.

name Specifies the name of the resource.

value Specifies the value of the resource if it will fit in an `XtArgVal` or the address.

The `XtSetArg` function is usually used in a highly stylized manner to minimize the probability of making a mistake; for example:

```

Arg args[20];
int n;

n = 0;
XtSetArg(args[n], XtNheight, 100);      n + +;
XtSetArg(args[n], XtNwidth, 200);      n + +;
XtSetValues(widget, args, n);

```

Alternatively, an application can statically declare the argument list and use `XtNumber`:

```

static Args args[] = {
    {XtNheight, (XtArgVal) 100},
    {XtNwidth, (XtArgVal) 200},
};
XtSetValues(Widget, args, XtNumber(args));

```

Note that you should not use auto-increment or auto-decrement within the first argument to `XtSetArg`. `XtSetArg` can be implemented as a macro that dereferences the first argument twice.

To merge two `ArgList` structures, use `XtMergeArgLists`.

```

ArgList XtMergeArgLists(args1, num_args1, args2, num_args2)
    ArgList args1;
    Cardinal num_args1;
    ArgList args2;
    Cardinal num_args2;

```

args1 Specifies the first `ArgList`.
num_args1 Specifies the number of arguments in the first argument list.
args2 Specifies the second `ArgList`.
num_args2 Specifies the number of arguments in the second argument list.

The `XtMergeArgLists` function allocates enough storage to hold the combined `ArgList` structures and copies them into it. Note that it does not check for duplicate entries. When it is no longer needed, free the returned storage by using `XtFree`.

2.4.2 Creating a Widget Instance

To create an instance of a widget, use `XtCreateWidget`.

Widget XtCreateWidget(*name*, *widget_class*, *parent*, *args*,
 num_args)

String *name*;
WidgetClass *widget_class*;
Widget *parent*;
ArgList *args*;
Cardinal *num_args*;

name Specifies the resource name for the created widget, which is used for retrieving resources and, for that reason, should not be the same as any other widget that is a child of same parent.

widget_class Specifies the widget class pointer for the created widget.

parent Specifies the parent widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in the argument list.

The XtCreateWidget function performs much of the boilerplate operations of widget creation:

- Checks to see if the `class_initialize` procedure has been called for this class and for all superclasses and, if not, calls those necessary in a superclass-to-subclass order.
- Allocates memory for the widget instance.
- If the parent is a subclass of `constraintWidgetClass`, it allocates memory for the parent's constraints and stores the address of this memory into the constraints field.
- Initializes the core nonresource data fields (for example, parent and visible).
- Initializes the resource fields (for example, `background_pixel`) by using the resource lists specified for this class and all superclasses.
- If the parent is a subclass of `constraintWidgetClass`, it initializes the resource fields of the constraints record by using the constraint resource list specified for the parent's class and all superclasses up to `constraintWidgetClass`.
- Calls the initialize procedures for the widget by starting at the Core initialize procedure on down to the widget's initialize procedure.
- If the parent is a subclass of `compositeWidgetClass`, it puts the widget into its parent's children list by calling its parent's `insert_child` procedure. For further information, see Section 3.5.
- If the parent is a subclass of `constraintWidgetClass`, it calls the constraint initialize procedures, starting at `constraintWidgetClass` on down to the parent's constraint initialize procedure.

Note that you can determine the number of arguments in an argument list by using the `XtNumber` macro. For further information, see Section 11.1. (See also `XtCreateManagedWidget`.)

2.4.3 Creating an Application Shell Instance

An application can have multiple top-level widgets, which can potentially be on many different screens. An application uses `XtAppCreateShell` if it needs to have several independent windows. The `XtAppCreateShell` function creates a top-level widget that is the root of a widget tree.

```
Widget XtAppCreateShell(application_name, application_class,  
                        widget_class, display, args,  
                        num_args)
```

```
String application_name;  
String application_class;  
WidgetClass widget_class;  
Display *display;  
ArgList args;  
Cardinal num_args;
```

application_name

Specifies the name of the application instance. If `application_name` is `NULL`, the application name passed to `XtDisplayInitialize` is used.

application_class

Specifies the class name of this application.

widget_class Specifies the widget class that the application top-level widget should be (normally, `applicationShellWidgetClass`).

display Specifies the display from which to get the resources.

args Specifies the argument list in which to set in the `WM_COMMAND` property.

num_args Specifies the number of arguments in the argument list.

The `XtAppCreateShell` function saves the specified application name and application class for qualifying all widget resource specifiers. The application name and application class are used as the left-most components in all widget resource names for this application. `XtAppCreateShell` should be used to create a new logical application within a program or to create a shell on another display. In the first case, it allows the specification of a new root in the resource hierarchy. In the second case, it uses the resource database associated with the other display.

Note that the widget returned by `XtAppCreateShell` has the `WM_COMMAND` property set for session managers (see Chapter 4).

To create multiple top-level shells within a single (logical) application, you can use one of two methods:

- Designate one shell as the real top-level shell and create the others as pop-up children of it by using `XtCreatePopupShell`.
- Have all shells as pop-up children of an unrealized top-level shell.

The first method, which is best used when there is a clear choice for what is the main window, leads to resource specifications like the following:

```
xmail.geometry:...      (the main window)
xmail.read.geometry:... (the read window)
xmail.compose.geometry:... (the compose window)
```

The second method, which is best if there is no main window, leads to resource specifications like the following:

```
xmail.headers.geometry:... (the headers window)
xmail.read.geometry:... (the read window)
xmail.compose.geometry:... (the compose window)
```

2.4.4 Widget Instance Initialization: the initialize Procedure

The initialize procedure pointer in a widget class is of type `XtInitProc`:

```
typedef void (*XtInitProc)(Widget, Widget);
    Widget request;
    Widget new;
```

request Specifies the widget with resource values as requested by the argument list, the resource database, and the widget defaults.

new Specifies a widget with the new values, both resource and nonresource, that are actually allowed.

An initialization procedure performs the following:

- Allocates space for and copies any resources that are referenced by address. For example, if a widget has a field that is a `String` it cannot depend on the characters at that address remaining constant but must dynamically allocate space for the string and copy it to the new space. (Note that you should not allocate space for or copy callback lists.)
- Computes values for unspecified resource fields. For example, if `width` and `height` are zero, the widget should compute an appropriate width and height based on other resources. This is the only time that a

widget should ever directly assign its own width and height.

- Computes values for uninitialized nonresource fields that are derived from resource fields. For example, graphics contexts (GCs) that the widget uses are derived from resources like background, foreground, and font.

An initialization procedure also can check certain fields for internal consistency. For example, it makes no sense to specify a color map for a depth that does not support that color map.

Initialization procedures are called in superclass-to-subclass order. Most of the initialization code for a specific widget class deals with fields defined in that class and not with fields defined in its superclasses.

If a subclass does not need an initialization procedure because it does not need to perform any of the above operations, it can specify NULL for the initialize field in the class record.

Sometimes a subclass may want to overwrite values filled in by its superclass. In particular, size calculations of a superclass are often incorrect for a subclass and in this case, the subclass must modify or recalculate fields declared and computed by its superclass.

As an example, a subclass can visually surround its superclass display. In this case, the width and height calculated by the superclass initialize procedure are too small and need to be incremented by the size of the surround. The subclass needs to know if its superclass's size was calculated by the superclass or was specified explicitly. All widgets must place themselves into whatever size is explicitly given, but they should compute a reasonable size if no size is requested.

The request and new arguments provide the necessary information for how a subclass knows the difference between a specified size and a size computed by a superclass. The request widget is the widget as originally requested. The new widget starts with the values in the request, but it has been updated by all superclass initialization procedures called so far. A subclass initialize procedure can compare these two to resolve any potential conflicts.

In the above example, the subclass with the visual surround can see if the width and height in the request widget are zero. If so, it adds its surround size to the width and height fields in the new widget. If not, it must make do with the size originally specified.

The new widget will become the actual widget instance record. Therefore, the initialization procedure should do all its work on the new widget (the request widget should never be modified), and if it needs to call any routines that operate on a widget, it should specify new as the widget instance.

2.4.5 Constraint Widget Instance Initialization: the `constraint_initialize` Procedure

The `constraint_initialize` procedure pointer is of type `XtInitProc`. The values passed to the parent constraint initialization procedure are the same as those passed to the child's class widget initialization procedure.

The constraint initialization procedure should compute any constraint fields derived from constraint resources. It can make further changes to the widget to make the widget conform to the specified constraints, for example, changing the widget's size or position.

If a constraint class does not need a constraint initialization procedure, it can specify `NULL` for the `initialize` field of the `ConstraintClassPart` in the class record.

2.4.6 Nonwidget Data Initialization: the `initialize_hook` Procedure

The `initialize_hook` procedure pointer is of type `XtArgsProc`:

```
typedef void (*XtArgsProc)(Widget, ArgList,
                           Cardinal *);

Widget w;
ArgList args;
Cardinal *num_args;
```

w Specifies the widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in the argument list.

If this procedure is not `NULL`, it is called immediately after the corresponding `initialize` procedure or in its place if the `initialize` procedure is `NULL`.

The `initialize_hook` procedure allows a widget instance to initialize nonwidget data using information from the specified argument list. For example, the `Text` widget has subparts that are not widgets, yet these subparts have resources that can be specified by means of the resource file or an argument list. See also Section 9.4.

2.5 Realizing Widgets

To realize a widget instance, use `XtRealizeWidget`.

```
void XtRealizeWidget(w)
    Widget w;
```

w Specifies the widget.

If the widget is already realized, `XtRealizeWidget` simply returns. Otherwise, it performs the following:

- Binds all action names in the widget's translation table to procedures (see Section 10.1.2).
- Makes a post-order traversal of the widget tree rooted at the specified widget and calls the `change_managed` procedure of each composite widget that has one or more managed children.
- Constructs an `XSetWindowAttributes` structure filled in with information derived from the Core widget fields and calls the `realize` procedure for the widget, which adds any widget-specific attributes and creates the X window.
- If the widget is not a subclass of `compositeWidgetClass`, `XtRealizeWidget` returns; otherwise, it continues and performs the following:
 - Descends recursively to each of the widget's managed children and calls the `realize` procedures. Primitive widgets that instantiate children are responsible for realizing those children themselves.
 - Maps all of the managed children windows that have `mapped_when_managed` `True`. (If a widget is managed but `mapped_when_managed` is `False`, the widget is allocated visual space but is not displayed. Some people seem to like this to indicate certain states.)

If the widget is a top-level shell widget (that is, it has no parent), and `mapped_when_managed` is `True`, `XtRealizeWidget` maps the widget window.

`XtCreateWidget`, `XtRealizeWidget`, `XtManageChildren`, `XtUnmanageChildren`, and `XtDestroyWidget` maintain the following invariants:

- If a widget is realized, then all its managed children are realized.
- If a widget is realized, then all its managed children that are also `mapped_when_managed` are mapped.

All Intrinsic functions and all widget routines should work with either realized or unrealized widgets.

To check whether or not a widget has been realized, use `XtIsRealized`.

```
Boolean XtIsRealized(w)
    Widget w;
```

w Specifies the widget.

The `XtIsRealized` function returns `True` if the widget has been realized, that is, if the widget has a nonzero X window ID.

Some widget procedures (for example, `set_values`) might wish to operate differently after the widget has been realized.

2.5.1 Widget Instance Window Creation: the realize Procedure

The `realize` procedure pointer in a widget class is of type `XtRealizeProc`:

```
typedef void (*XtRealizeProc)(Widget, XtValueMask *,
                               XSetWindowAttributes *);
    Widget w;
    XtValueMask *value_mask;
    XSetWindowAttributes *attributes;
```

w Specifies the widget.

value_mask Specifies which fields in the attributes structure to use.

attributes Specifies the window attributes to use in the `XCreateWindow` call.

The `realize` procedure must create the widget's window.

The generic `XtRealizeWidget` function fills in a mask and a corresponding `XSetWindowAttributes` structure. It sets the following fields based on information in the widget `Core` structure:

- The `background_pixmap` (or `background_pixel` if `background_pixmap` is `NULL`) is filled in from the corresponding field.
- The `border_pixmap` (or `border_pixel` if `border_pixmap` is `NULL`) is filled in from the corresponding field.
- The `event_mask` is filled in based on the event handlers registered, the event translations specified, whether `expose` is non-`NULL`, and whether `visible_interest` is `True`.
- The `bit_gravity` is set to `NorthWestGravity` if the `expose` field is `NULL`.
- The `do_not_propagate_mask` is set to propagate all pointer and keyboard events up the window tree. A composite widget can implement functionality caused by an event anywhere inside it (including on top of children widgets) as long as children do not specify a translation for the event.

All other fields in attributes (and the corresponding bits in `value_mask`) can be set by the realize procedure.

Note that because realize is not a chained operation, the widget class realize procedure must update the `XSetWindowAttributes` structure with all the appropriate fields from non-Core superclasses.

A widget class can inherit its realize procedure from its superclass during class initialization. The realize procedure defined for Core calls `XtCreateWindow` with the passed `value_mask` and attributes and with `windowClass` and `visual` set to `CopyFromParent`. Both `CompositeWidgetClass` and `ConstraintWidgetClass` inherit this realize procedure, and most new widget subclasses can do the same (see Section 1.4.9).

The most common noninherited realize procedures set `bit_gravity` in the mask and attributes to the appropriate value and then create the window. For example, depending on its justification, `Label` sets `bit_gravity` to `WestGravity`, `CenterGravity`, or `EastGravity`. Consequently, shrinking it just moves the bits appropriately, and no `Expose` event is needed for repainting.

If a composite widget's children should be realized in a particular order (typically to control the stacking order), it should call `XtRealizeWidget` on its children itself in the appropriate order from within its own realize procedure.

Widgets that have children and that are not a subclass of `compositeWidgetClass` are responsible for calling `XtRealizeWidget` on their children, usually from within the realize procedure.

2.5.2 Window Creation Convenience Routine

Rather than call the Xlib `XCreateWindow` function explicitly, a realize procedure should call the Intrinsics analog `XtCreateWindow`, which simplifies the creation of windows for widgets.

```
void XtCreateWindow(w, window_class, visual, value_mask,  
                   attributes)  
  
    Widget w;  
    unsigned int window_class;  
    Visual *visual;  
    XtValueMask value_mask;  
    XSetWindowAttributes *attributes;
```

w Specifies the widget that is used to set the x,y coordinates and so on.

window_class Specifies the Xlib window class (for example, `InputOutput`, `InputOnly`, or `CopyFromParent`).

visual Specifies the visual type (usually CopyFromParent).
value_mask Specifies which attribute fields to use.
attributes Specifies the window attributes to use in the XCreateWindow call.

The XtCreateWindow function calls the Xlib XCreateWindow function with values from the widget structure and the passed parameters. Then, it assigns the created window to the widget's window field.

XtCreateWindow evaluates the following fields of the Core widget structure:

- depth
- screen
- parent -> core.window
- x
- y
- width
- height
- border_width

2.6 Obtaining Window Information from a Widget

The Core widget definition contains the screen and window IDs. The window field may be NULL for a while (see Sections 2.4 and 2.5).

The display pointer, the parent widget, screen pointer, and window of a widget are available to the widget writer by means of macros and to the application writer by means of functions.

```
Display *XtDisplay(w)
Widget w;
```

w Specifies the widget.

XtDisplay returns the display pointer for the specified widget.

```
Widget XtParent(w)
Widget w;
```

w Specifies the widget.

XtParent returns the parent widget for the specified widget.

```
Screen *XtScreen(w)
    Widget w;
```

w Specifies the widget.

XtScreen returns the screen pointer for the specified widget.

```
Window XtWindow(w)
    Widget w;
```

w Specifies the widget.

XtWindow returns the window of the specified widget.

Several window attributes are locally cached in the widget. Thus, they can be set by the resource manager and `XtSetValues` as well as used by routines that derive structures from these values (for example, depth for deriving pixmaps, `background_pixel` for deriving GCs, and so on) or in the `XtCreateWindow` call.

The `x`, `y`, `width`, `height`, and `border_width` window attributes are available to geometry managers. These fields are maintained synchronously inside the XUI Toolkit. When an `XConfigureWindow` is issued on the widget's window (on request of its parent), these values are updated immediately rather than sometime later when the server generates a `ConfigureNotify` event. (In fact, most widgets do not have `SubstructureNotify` turned on.) This ensures that all geometry calculations are based on the internally consistent toolkit world, rather than on either an inconsistent world updated by asynchronous `ConfigureNotify` events or a consistent but slow world in which geometry managers ask the server for window sizes whenever they need to lay out their managed children (see Chapter 6).

2.6.1 Unrealizing Widgets

To destroy the windows associated with a widget and its descendants, use `XtUnrealizeWidget`.

```
void XtUnrealizeWidget(w)
    Widget w;
```

w Specifies the widget.

The `XtUnrealizeWidget` function destroys the windows of an existing widget and all of its children (recursively down the widget tree). To recreate the windows at a later time, call `XtRealizeWidget` again. If the widget was managed, it will be unmanaged automatically before its window is freed.

2.7 Destroying Widgets

The Ininsics provide support to:

- Destroy all the pop-up children of the widget being destroyed and destroy all children of composite widgets
- Remove (and unmap) the widget from its parent
- Call the callback procedures that have been registered to trigger when the widget is destroyed
- Minimize the number of things a widget has to deallocate when destroyed
- Minimize the number of XDestroyWindow calls

To destroy a widget instance, use XtDestroyWidget.

```
void XtDestroyWidget(w)
    Widget w;
```

w Specifies the widget.

The XtDestroyWidget function provides the only method of destroying a widget, including widgets that need to destroy themselves. It can be called at any time, including from an application callback routine of the widget being destroyed. This requires a two-phase destroy process in order to avoid dangling references to destroyed widgets.

In phase one, XtDestroyWidget performs the following:

- If the being_destroyed field of the widget is True, it returns immediately.
- Recursively descends the widget tree and sets the being_destroyed field to True for the widget and all children.
- Adds the widget to a list of widgets (the destroy list) that should be destroyed when it is safe to do so.

Entries on the destroy list satisfy the invariant that if w2 occurs after w1 on the destroy list then w2 is not a descendent of w1. (A descendant refers to both normal and pop-up children.)

Phase two occurs when all procedures that should execute as a result of the current event have been called (including all procedures registered with the event and translation managers), that is, when the current invocation of XtDispatchEvent is about to return or immediately if not in XtDispatchEvent.

In phase two, XtDestroyWidget performs the following on each entry in the destroy list:

- Calls the destroy callback procedures registered on the widget (and all descendants) in post-order (it calls children callbacks before parent callbacks).
- If the widget's parent is a subclass of `compositeWidgetClass` and if the parent is not being destroyed, it calls `XtUnmanageChild` on the widget and then calls the widget's parent's `delete_child` procedure (see Section 3.4).
- If the widget's parent is a subclass of `constraintWidgetClass`, it calls the constraint destroy procedure for the parent, then the parent's superclass, until finally it calls the constraint destroy procedure for `constraintWidgetClass`.
- Calls the destroy methods for the widget (and all descendants) in post-order. For each such widget, it calls the destroy procedure declared in the widget class, then the destroy procedure declared in its superclass, until finally it calls the destroy procedure declared in the Core class record.
- Calls `XDestroyWindow` if the widget is realized (that is, has an X window). The server recursively destroys all descendant windows.
- Recursively descends the tree and deallocates all pop-up widgets, constraint records, callback lists and, if the widget is a subclass of `compositeWidgetClass`, children.

2.7.1 Adding and Removing Destroy Callbacks

When an application needs to perform additional processing during the destruction of a widget, it should register a destroy callback procedure for the widget. The destroy callback procedures use the mechanism described in Chapter 8. The destroy callback list is identified by the resource name `XtNdestroyCallback`.

For example, the following adds an application-supplied destroy callback procedure `ClientDestroy` with client data to a widget by calling `XtAddCallback`.

```
XtAddCallback(w, XtNdestroyCallback, ClientDestroy,
              client_data)
```

Similarly, the following removes the application-supplied destroy callback procedure `ClientDestroy` by calling `XtRemoveCallback`.


```
XtRemoveCallback(w, XtNdestroyCallback, ClientDestroy,  
                client_data)
```

The *ClientDestroy* argument is of type `XtCallbackProc`:

```
typedef void (*XtCallbackProc)(Widget, caddr_t, caddr_t);
```

For further information, see Section 8.1.

2.7.2 Dynamic Data Deallocation: the destroy Procedure

The destroy procedure pointer in the `CoreClassPart` structure is of type `XtWidgetProc`:

```
typedef void (*XtWidgetProc)(Widget);
```

The destroy procedures are called in subclass-to-superclass order. Therefore, a widget's destroy procedure only should deallocate storage that is specific to the subclass and should not bother with the storage allocated by any of its superclasses. The destroy procedure should only deallocate resources that have been explicitly created by the subclass. Any resource that was obtained from the resource database or was passed in in an argument list was not created by the widget and, therefore, should not be destroyed by it. If a widget does not need to deallocate any storage, the destroy procedure entry in its widget class record can be `NULL`.

Deallocating storage includes but is not limited to:

- Calling `XtFree` on dynamic storage allocated with `XtMalloc`, `XtCalloc`, and so on
- Calling `XFreePixmap` on pixmaps created with direct X calls
- Calling `XtDestroyGC` on GCs allocated with `XtGetGC`
- Calling `XFreeGC` on GCs allocated with direct X calls
- Calling `XtRemoveEventHandler` on event handlers added with `XtAddEventHandler`
- Calling `XtRemoveTimeOut` on timers created with `XtAppAddTimeOut`
- Calling `XtDestroyWidget` for each child if the widget has children and is not a subclass of `compositeWidgetClass`

2.7.3 Dynamic Constraint Data Deallocation: the constraint destroy Procedure

The constraint destroy procedure identified in the `ConstraintClassPart` structure is called for a widget whose parent is a subclass of `constraintWidgetClass`. This constraint destroy procedure pointer is of type `XtWidgetProc`. The constraint destroy procedures are called in subclass-to-

superclass order, starting at the widget's parent and ending at `constraintWidgetClass`. Therefore, a parent's constraint destroy procedure only should deallocate storage that is specific to the constraint subclass and not the storage allocated by any of its superclasses.

If a parent does not need to deallocate any constraint storage, the constraint destroy procedure entry in its class record can be `NULL`.

2.8 Exiting from an Application

All XUI Toolkit applications should terminate by calling `XtDestroyApplicationContext` and then exiting using the standard method for their operating system (typically, by calling `exit` for UNIX-based systems). The quickest way to make the windows disappear while exiting is to call `XtUnmapWidget` on each top-level shell widget. The XUI Toolkit has no resources beyond those in the program image, and the X server will free its resources when its connection to the application is broken.

Composite Widgets and Their Children 3

Composite widgets (widgets that are a subclass of `compositeWidgetClass`) can have an arbitrary number of children. Consequently, they are responsible for much more than primitive widgets. Their responsibilities (either implemented directly by the widget class or indirectly by Intrinsic functions) include:

- Overall management of children from creation to destruction
- Destruction of descendants when the composite widget is destroyed
- Physical arrangement (geometry management) of a displayable subset of children (that is, the managed children)
- Mapping and unmapping of a subset of the managed children

Overall management is handled by the generic procedures `XtCreateWidget` and `XtDestroyWidget`. `XtCreateWidget` adds children to their parent by calling the parent's `insert_child` procedure. `XtDestroyWidget` removes children from their parent by calling the parent's `delete_child` procedure and ensures that all children of a destroyed composite widget also get destroyed.

Only a subset of the total number of children is actually managed by the geometry manager and, hence, possibly visible. For example, a multibuffer composite editor widget might allocate one child widget for each file buffer, but it only might display a small number of the existing buffers. Windows that are in this displayable subset are called managed windows and enter into geometry manager calculations. The other children are called unmanaged windows and, by definition, are not mapped.

Children are added to and removed from the managed set by using `XtManageChild`, `XtManageChildren`, `XtUnmanageChild`, and `XtUnmanageChildren`, which notify the parent to recalculate the physical layout of its children by calling the parent's `change_managed` procedure. The `XtCreateManagedWidget` convenience function calls `XtCreateWidget` and `XtManageChild` on the result.

Most managed children are mapped, but some widgets can be in a state where they take up physical space but do not show anything. Managed widgets are not mapped automatically if their `map_when_managed` field is `False`. The default is `True` and is changed by using `XtSetMappedWhenManaged`.

Each composite widget class has a geometry manager, which is responsible for figuring out where the managed children should appear within the composite widget's window. Geometry management techniques fall into four classes:

- **Fixed boxes**
Fixed boxes have a fixed number of children that are created by the parent. All of these children are managed, and none ever make geometry manager requests.
- **Homogeneous boxes**
Homogeneous boxes treat all children equally and apply the same geometry constraints to each child. Many clients insert and delete widgets freely.
- **Heterogeneous boxes**
Heterogeneous boxes have a specific location where each child is placed. This location usually is not specified in pixels, because the window may be resized, but is expressed rather in terms of the relationship between a child and the parent or between the child and other specific children. Heterogeneous boxes are usually subclasses of `Constraint`.
- **Shell boxes**
Shell boxes have only one child, which is exactly the size of the shell. The geometry manager must communicate with the window manager if it exists, and the box must also accept `ConfigureNotify` events when the window size is changed by the window manager.

3.1 Verifying the Class of a Composite Widget

To test if a given widget is a subclass of `Composite`, use `XtIsComposite`.

```
Boolean XtIsComposite(w)  
    Widget w;
```

w Specifies the widget.

The `XtIsComposite` function is a convenience function that is equivalent to `XtIsSubclass` with `compositeWidgetClass` specified.

3.2 Addition of Children to a Composite Widget: the `insert_child` Procedure

To add a child to the parent's list of children, the `XtCreateWidget` function calls the parent's class routine `insert_child`. The `insert_child` procedure

pointer in a composite widget is of type `XtWidgetProc`:

```
typedef void (*XtWidgetProc)(Widget);
```

Most composite widgets inherit their superclass's operation. Composite's `insert_child` routine calls the `insert_position` procedure and inserts the child at the specified position.

Some composite widgets define their own `insert_child` routine so that they can order their children in some convenient way, create companion controller widgets for a new widget, or limit the number or type of their children widgets.

If there is not enough room to insert a new child in the children array (that is, `num_children = num_slots`), the `insert_child` procedure must first reallocate the array and update `num_slots`. The `insert_child` procedure then places the child wherever it wants and increments the `num_children` field.

3.3 Insertion Order of Children: the `insert_position` Procedure

Instances of composite widgets need to specify about the order in which their children are kept. For example, an application may want a set of command buttons in some logical order grouped by function, and it may want buttons that represent file names to be kept in alphabetical order.

The `insert_position` procedure pointer in a composite widget instance is of type `XtOrderProc`:

```
typedef Cardinal (*XtOrderProc)(Widget);  
                Widget w;
```

w Specifies the widget.

Composite widgets that allow clients to order their children (usually homogeneous boxes) can call their widget instance's `insert_position` procedure from the class's `insert_child` procedure to determine where a new child should go in its children array. Thus, a client of a composite class can apply different sorting criteria to widget instances of the class, passing in a different `insert_position` procedure when it creates each composite widget instance.

The return value of the `insert_position` procedure indicates how many children should go before the widget. Returning zero indicates that the widget should go before all other children, and returning `num_children` indicates that it should go after all other children. The default `insert_position` function returns `num_children` and can be overridden by a

specific composite widget's resource list or by the argument list provided when the composite widget is created.

3.4 Deletion of Children: the `delete_child` Procedure

To remove the child from the parent's children array, the `XtDestroyWidget` function eventually causes a call to the composite parent's class `delete_child` procedure. The `delete_child` procedure pointer is of type `XtWidgetProc`:

```
typedef void (*XtWidgetProc)(Widget);
```

Most widgets inherit the `delete_child` procedure from their superclass. Composite widgets that create companion widgets define their own `delete_child` procedure to remove these companion widgets.

3.5 Adding and Removing Children from the Managed Set

The Intrinsic provide a set of generic routines to permit the addition of widgets to or the removal of widgets from a composite widget's managed set. These generic routines eventually call the widget's `change_managed` procedure. The `change_managed` procedure pointer is of type `XtWidgetProc`.

3.5.1 Managing Children

To add a list of widgets to the geometry-managed (and, hence, displayable) subset of its composite parent widget, the application must first create the widgets (`XtCreateWidget`) and then call `XtManageChildren`.

```
typedef Widget *WidgetList;

void XtManageChildren(children, num_children)
    WidgetList children;
    Cardinal num_children;
```

children Specifies a list of child widgets.

num_children Specifies the number of children.

The `XtManageChildren` function performs the following:

- Issues an error if the children do not all have the same parent or if the parent is not a subclass of `compositeWidgetClass`.
- Returns immediately if the common parent is being destroyed; otherwise, for each unique child on the list, `XtManageChildren` ignores the child if it already is managed or is being destroyed and marks it if not.

- If the parent is realized and after all children have been marked, it makes some of the newly managed children viewable:
 - Calls the `change_managed` routine of the widgets' parent.
 - Calls `XtRealizeWidget` on each previously unmanaged child that is unrealized.
 - Maps each previously unmanaged child that has `map_when_managed` `True`.

Managing children is independent of the ordering of children and independent of creating and deleting children. The layout routine of the parent should consider children whose `managed` field is `True` and should ignore all other children. Note that some composite widgets, especially fixed boxes, call `XtManageChild` from their `insert_child` procedure.

If the parent widget is realized, its `change_managed` procedure is called to notify it that its set of managed children has changed. The parent can reposition and resize any of its children. It moves each child as needed by calling `XtMoveWidget`, which first updates the `x` and `y` fields and then calls `XMoveWindow` if the widget is realized.

If the composite widget wishes to change the size or border width of any of its children, it calls `XtResizeWidget`, which first updates the `Core` fields and then calls the Xlib `XConfigureWindow` function if the widget is realized.

To add a single child to a parent widget's list of managed children, first create the child widget (`XtCreateWidget`) and then use `XtManageChild`.

```
void XtManageChild(child)
    Widget child;
```

child Specifies the child.

The `XtManageChild` function constructs a `WidgetList` of length one and calls `XtManageChildren`.

To create and manage a child widget in a single procedure, use `XtCreateManagedWidget`.

```
Widget XtCreateManagedWidget(name, widget_class, parent,
                               args, num_args)
    String name;
    WidgetClass widget_class;
    Widget parent;
    ArgList args;
    Cardinal num_args;
```


name Specifies the text name for the created widget.
widget_class Specifies the widget class pointer for the created widget.
parent Specifies the parent widget.
args Specifies the argument list to override the resource defaults.
num_args Specifies the number of arguments in the argument list.

The `XtCreateManagedWidget` function is a convenience routine that calls `XtCreateWidget` and `XtManageChild`.

3.5.2 Unmanaging Children

To remove a list of children from a parent widget's managed list, use `XtUnmanageChildren`.

```
void XtUnmanageChildren(children, num_children)
    WidgetList children;
    Cardinal num_children;
```

children Specifies a list of child widgets.
num_children Specifies the number of children.

The `XtUnmanageChildren` function performs the following:

- Issues an error if the children do not all have the same parent or if the parent is not a subclass of `compositeWidgetClass`.
- Returns immediately if the common parent is being destroyed; otherwise, for each unique child on the list, `XtUnmanageChildren` performs the following:
 - Ignores the child if it already is unmanaged or is being destroyed and marks it if not.
 - If the child is realized, it makes it nonvisible by unmapping it.
- Calls the `change_managed` routine of the widgets' parent after all children have been marked if the parent is realized.

`XtUnmanageChildren` does not destroy the children widgets. Removing widgets from a parent's managed set is often a temporary banishment, and, some time later, you may manage the children again. To destroy widgets entirely, see Section 2.7.

To remove a single child from its parent's managed set, use `XtUnmanageChild`.

```
void XtUnmanageChild(child)
    Widget child;
```

child Specifies the child.

The `XtUnmanageChild` function constructs a widget list of length one and calls `XtUnmanageChildren`.

These generic functions are low-level routines that are used by generic composite widget building routines. In addition, composite widgets can provide widget-specific, high-level convenience procedures to let applications create and manage children more easily.

3.5.3 Determining if a Widget Is Managed

To determine the managed state of a given child widget, use `XtIsManaged`.

```
Boolean XtIsManaged(w)
    Widget w;
```

w Specifies the widget.

The `XtIsManaged` macro (for widget programmers) or function (for application programmers) returns `True` if the specified child widget is managed or `False` if it is not.

3.6 Controlling When Widgets Get Mapped

A widget is normally mapped if it is managed. However, this behavior can be overridden by setting the `XtNmappedWhenManaged` resource for the widget when it is created or by setting the `map_when_managed` field to `False`.

To change the value of a given widget's `map_when_managed` field, use `XtSetMappedWhenManaged`.

```
void XtSetMappedWhenManaged(w, map_when_managed)
    Widget w;
    Boolean map_when_managed;
```

w Specifies the widget.

map_when_managed

Specifies a Boolean value that indicates the new value of the `map_when_managed` field.

If the widget is realized and managed and if the new value of `map_when_managed` is `True`, `XtSetMappedWhenManaged` maps the window. If the widget is realized and managed and if the new value of

`map_when_managed` is `False`, it unmaps the window.

`XtSetMappedWhenManaged` is a convenience function that is equivalent to (but slightly faster than) calling `XtSetValues` and setting the new value for the `mappedWhenManaged` resource. As an alternative to using `XtSetMappedWhenManaged` to control mapping, a client may set `mapped_when_managed` to `False` and use `XtMapWidget` and `XtUnmapWidget` explicitly.

To map a widget explicitly, use `XtMapWidget`.

```
XtMapWidget(w)
    Widget w;
```

w Specifies the widget.

To unmap a widget explicitly, use `XtUnmapWidget`.

```
XtUnmapWidget(w)
    Widget w;
```

w Specifies the widget.

3.7 Constrained Composite Widgets

Constraint widgets are a subclass of `compositeWidgetClass`. Their name is derived from the fact that they may manage the geometry of their children based on constraints associated with each child. These constraints can be as simple as the maximum width and height the parent will allow the child to occupy or can be as complicated as how other children should change if this child is moved or resized. Constraint widgets let a parent define resources that are supplied for their children. For example, if the Constraint parent defines the maximum sizes for its children, these new size resources are retrieved for each child as if they were resources that were defined by the child widget. Accordingly, constraint resources may be included in the argument list or resource file just like any other resource for the child.

Constraint widgets have all the responsibilities of normal composite widgets and, in addition, must process and act upon the constraint information associated with each of their children.

To make it easy for widgets and the Intrinsic to keep track of the constraints associated with a child, every widget has a `constraints` field, which is the address of a parent-specific structure that contains constraint information about the child. If a child's parent is not a subclass of `constraintWidgetClass`, then the child's `constraints` field is `NULL`.

Subclasses of a Constraint widget can add additional constraint fields to their superclass. To allow this, widget writers should define the constraint records in their private .h file by using the same conventions as used for widget records. For example, a widget that needs to maintain a maximum width and height for each child might define its constraint record as follows:

```
typedef struct {
    Dimension max_width, max_height;
} MaxConstraintPart;
```

```
typedef struct {
    MaxConstraintPart max;
} MaxConstraintRecord, *MaxConstraint;
```

A subclass of this widget that also needs to maintain a minimum size would define its constraint record as follows:

```
typedef struct {
    Dimension min_width, min_height;
} MinConstraintPart;
```

```
typedef struct {
    MaxConstraintPart max;
    MinConstraintPart min;
} MaxMinConstraintRecord, *MaxMinConstraint;
```

Constraints are allocated, initialized, deallocated, and otherwise maintained insofar as possible by the Intrinsics. The constraint class record part has several entries that facilitate this. All entries in ConstraintClassPart are information and procedures that are defined and implemented by the parent, but they are called whenever actions are performed on the parent's children.

The XtCreateWidget function uses the constraint_size field to allocate a constraint record when a child is created. The constraint_size field gives the number of bytes occupied by a constraint record. XtCreateWidget also uses the constraint resources to fill in resource fields in the constraint record associated with a child. It then calls the constraint initialize procedure so that the parent can compute constraint fields that are derived from constraint resources and can possibly move or resize the child to conform to the given constraints.

The XtGetValues and XtSetValues functions use the constraint resources to get the values or set the values of constraints associated with a child. XtSetValues then calls the constraint set_values procedures so that a parent can recompute derived constraint fields and move or resize the child as appropriate.

The `XtDestroyWidget` function calls the constraint destroy procedure to deallocate any dynamic storage associated with a constraint record. The constraint record itself must not be deallocated by the constraint destroy procedure; `XtDestroyWidget` does this automatically.

Shell Widgets 4

Shell widgets hold an application's top-level widgets to allow them to communicate with the window manager. Shells have been designed to be as nearly invisible as possible. Clients have to create them, but they should never have to worry about their sizes.

If a shell widget is resized from the outside (typically by a window manager), the shell widget also resizes its child widget automatically. Similarly, if the shell's child widget needs to change size, it can make a geometry request to the shell, and the shell negotiates the size change with the outer environment. Clients should never attempt to change the size of their shells directly.

The four types of public shells are:

- | | |
|-------------------------|--|
| OverrideShell | Used for shell windows that completely bypass the window manager (for example, pop-up menu shells). |
| TransientShell | Used for shell windows that can be manipulated by the window manager but are not allowed to be iconified separately (for example, Dialog boxes that make no sense without their associated application). They are iconified by the window manager only if the main application shell is iconified. |
| TopLevelShell | Used for normal top-level windows (for example, any additional top-level widgets an application needs). |
| ApplicationShell | Used as the main top-level window for an application. An application should only have more than one <code>ApplicationShell</code> if it implements multiple logical applications. |

4.1 Shell Widget Definitions

Widgets negotiate their size and position with their parent widget, that is, the widget that directly contains them. Widgets at the top of the hierarchy do not have parent widgets. Instead, they must deal with the outside world. To provide for this, each top-level widget is encapsulated in a special widget, called a Shell.

Shell widgets, a subclass of the Composite widget, encapsulate other widgets and can allow a widget to avoid the geometry clipping imposed by the parent/child window relationship. They also can provide a layer of communication with the window manager.

The seven different types of shells are:

Shell	Provides the base class for shell widgets and the fields needed for all types of shells. Shell is a direct subclass of compositeWidgetClass.
OverrideShell	Used for shell windows that completely bypass the window manager and is a subclass of Shell.
WMShell	Contains fields needed by the common window manager protocol and is a subclass of Shell.
VendorShell	Contains fields used by vendor-specific window managers and is a subclass of WMShell.
TransientShell	Used for shell windows that can be manipulated by the window manager but that are not allowed to be iconified and is a subclass of VendorShell.
TopLevelShell	Used for normal top level windows and is a subclass of VendorShell.
ApplicationShell	Used for an application's top-level window and is a subclass of TopLevelShell.

Note that the classes Shell, WMShell, and VendorShell are internal and should not be instantiated or subclassed. Only OverrideShell, TransientShell, TopLevelShell, and ApplicationShell are for public use.

4.1.1 ShellClassPart Definitions

None of the shell widget classes has any additional fields:

```
typedef struct { caddr_t extension; } ShellClassPart, OverrideShellClassPart,  
                WMShellClassPart, VendorShellClassPart, TransientShellClassPart,  
                TopLevelShellClassPart, ApplicationShellClassPart;
```

4-2 Shell Widgets

Shell widget classes have the (empty) shell fields immediately following the composite fields:

```
typedef struct _ShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
} ShellClassRec;

typedef struct _OverrideShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    OverrideShellClassPart override_shell_class;
} OverrideShellClassRec;

typedef struct _WMShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    WMShellClassPart wm_shell_class;
} WMShellClassRec;

typedef struct _VendorShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    WMShellClassPart wm_shell_class;
    VendorShellClassPart vendor_shell_class;
} VendorShellClassRec;

typedef struct _TransientShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    WMShellClassPart wm_shell_class;
    VendorShellClassPart vendor_shell_class;
    TransientShellClassPart transient_shell_class;
} TransientShellClassRec;

typedef struct _TopLevelShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    WMShellClassPart wm_shell_class;
    VendorShellClassPart vendor_shell_class;
    TopLevelShellClassPart top_level_shell_class;
}
```



```

} TopLevelShellClassRec;

typedef struct _ApplicationShellClassRec {
    CoreClassPart core_class;
    CompositeClassPart composite_class;
    ShellClassPart shell_class;
    WMShellClassPart wm_shell_class;
    VendorShellClassPart vendor_shell_class;
    TopLevelShellClassPart top_level_shell_class;
    ApplicationShellClassPart application_shell_class;
} ApplicationShellClassRec;

```

The predefined class records and pointers for shells are:

```

extern ShellClassRec shellClassRec;
extern OverrideShellClassRec overrideShellClassRec;
extern WMShellClassRec wmShellClassRec;
extern VendorShellClassRec vendorShellClassRec;
extern TransientShellClassRec transientShellClassRec;
extern TopLevelShellClassRec topLevelShellClassRec;
extern ApplicationShellClassRec applicationShellClassRec;

extern WidgetClass shellWidgetClass;
extern WidgetClass overrideShellWidgetClass;
extern WidgetClass wmShellWidgetClass;
extern WidgetClass vendorShellWidgetClass;
extern WidgetClass transientShellWidgetClass;
extern WidgetClass topLevelShellWidgetClass;
extern WidgetClass applicationShellWidgetClass;

```

The following opaque types and opaque variables are defined for generic operations on widgets that are a subclass of `ShellWidgetClass`:

Types	Variables
ShellWidget	shellWidgetClass
OverrideShellWidget	overrideShellWidgetClass
WMShellWidget	wmShellWidgetClass
VendorShellWidget	vendorShellWidgetClass
TransientShellWidget	transientShellWidgetClass
TopLevelShellWidget	topLevelShellWidgetClass
ApplicationShellWidget	applicationShellWidgetClass
ShellWidgetClass	
OverrideShellWidgetClass	
WMShellWidgetClass	
VendorShellWidgetClass	

Types**Variables**

TransientShellWidgetClass
TopLevelShellWidgetClass
ApplicationShellWidgetClass

4.1.2 ShellPart Definition

The various shells have the following additional fields defined in their widget records:

```
typedef struct {
    String geometry;
    XtCreatePopupChildProc create_popup_child_proc;
    XtGrabKind grab_kind;
    Boolean spring_loaded;
    Boolean popped_up;
    Boolean allow_shell_resize;
    Boolean client_specified;
    Boolean save_under;
    Boolean override_redirect;
    XtCallbackList popup_callback;
    XtCallbackList popdown_callback;
} ShellPart;

typedef struct { int empty; } OverrideShellPart;

typedef struct {
    String title;
    int wm_timeout;
    Boolean wait_for_wm;
    Boolean transient;
    XSizeHints size_hints;
    XWMHints wm_hints;
} WMShellPart;

typedef struct {
    int vendor_specific;
} VendorShellPart;

typedef struct { int empty; } TransientShellPart;

typedef struct {
    String icon_name;
```

```

        Boolean iconic;
    } TopLevelShellPart;

typedef struct {
    char *class;
    XrmClass xrm_class;
    int argc;
    char **argv;
} ApplicationShellPart;

```

The full definitions of the various shell widgets have shell fields following composite fields:

```

typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
} ShellRec, *ShellWidget;

typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    OverrideShellPart override;
} OverrideShellRec, *OverrideShellWidget;

typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    WMShellPart wm;
} WMShellRec, *WMShellWidget;

typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    WMShellPart wm;
    VendorShellPart vendor;
} VendorShellRec, *VendorShellWidget;

typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    WMShellPart wm;
    VendorShellPart vendor;
}

```

```

    TransientShellPart transient;
} TransientShellRec, *TransientShellWidget;
typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    WMShellPart wm;
    VendorShellPart vendor;
    TopLevelShellPart topLevel;
} TopLevelShellRec, *TopLevelShellWidget;
typedef struct {
    CorePart core;
    CompositePart composite;
    ShellPart shell;
    WMShellPart wm;
    VendorShellPart vendor;
    TopLevelShellPart topLevel;
    ApplicationShellPart application;
} ApplicationShellRec, *ApplicationShellWidget;

```

4.1.3 ShellPart Default Values

The default values for fields common to all classes of public shells (filled in by the Shell resource lists and the Shell initialize procedures) are:

Field	Default Value
geometry	NULL
create_popup_child_proc	NULL
grab_kind	(internal)
spring_loaded	(internal)
popped_up	(internal)
allow_shell_resize	False
client_specified	(internal)
save_under	True for OverrideShell and TransientShell, False otherwise
override_redirect	True for OverrideShell, False otherwise
popup_callback	NULL
popdown_callback	NULL

The geometry resource specifies the size and position and is usually done only from a command line or a defaults file. For further information, see the *Guide to the Xlib Library*. The `create_popup_child_proc` is called by the `XtPopup` procedure and is usually `NULL`. The `allow_shell_resize` field controls whether or not the widget contained by the shell is allowed to try to resize itself. If `allow_shell_resize` is `False`, any geometry requests always return `XtGeometryNo`. Setting `save_under` instructs the server to attempt to save the contents of windows obscured by the shell when it is mapped and to restore its contents automatically later. It is useful for pop-up menus. Setting `override_redirect` determines whether or not the shell window is visible to the window manager. If it is `True`, the window is immediately mapped without the manager's intervention. The `popup` and `popdown` callbacks are called during `XtPopup` and `XtPopdown`. For further information, see the *Guide to the Xlib Library*.

The default values for shell fields in `WMShell` and its subclasses are:

Field	Default Value
<code>title</code>	Icon name, if specified, otherwise the application's name
<code>wm_timeout</code>	Five seconds
<code>wait_for_wm</code>	<code>True</code>
<code>transient</code>	<code>True</code> for <code>TransientShell</code> , <code>False</code> otherwise
<code>min_width</code>	<code>None</code>
<code>min_height</code>	<code>None</code>
<code>max_width</code>	<code>None</code>
<code>max_height</code>	<code>None</code>
<code>width_inc</code>	<code>None</code>
<code>height_inc</code>	<code>None</code>
<code>min_aspect_x</code>	<code>None</code>
<code>min_aspect_y</code>	<code>None</code>
<code>max_aspect_x</code>	<code>None</code>
<code>max_aspect_y</code>	<code>None</code>
<code>input</code>	<code>False</code>
<code>initial_state</code>	<code>Normal</code>
<code>icon_pixmap</code>	<code>None</code>
<code>icon_window</code>	<code>None</code>
<code>icon_x</code>	<code>None</code>
<code>icon_y</code>	<code>None</code>
<code>icon_mask</code>	<code>None</code>
<code>window_group</code>	<code>None</code>

The title is a string to be displayed by the window manager. The `wm_timeout` resource limits the amount of time a shell is to wait for confirmation of a geometry request to the window manager. If none comes back within that time, the shell assumes the window manager is not functioning properly and sets `wait_for_wm` to be `False` (later events may reset this value). The `wait_for_wm` resource sets the initial state for this flag. When the flag is `False`, the shell does not wait for a response but relies on asynchronous notification. All other resources are for fields in the window manager hints and the window manager size hints. For further information, see the *Guide to the Xlib Library*.

TopLevel shells have the the following additional resources:

Field	Default Value
<code>icon_name</code>	Shell widget's name
<code>iconic</code>	<code>False</code>

The `icon_name` field is the string to display in the shell's icon, and the `iconic` field is an alternative way to set the `initialState` resource to indicate that a shell should be initially displayed as an icon.

Application shells have the following additional resources:

Field	Default Value
<code>argc</code>	0
<code>argv</code>	NULL

The `argc` and `argv` fields are used to initialize the standard property `WM_COMMAND`.

4.1.4 Digital's Vendor Shell Implementation

Digital's implementation of the `VendorShell` has the the following additional resources:

Field	Type	Default Value
iconify_pixmap	Pixmap	Unspecified
icon_box_x	Int	Unspecified
icon_box_y	Int	Unspecified
tiled	Bool	Unspecified
sticky	Bool	Unspecified
no_iconify	Bool	Unspecified
no_lower	Bool	Unspecified
no_resize	Bool	Unspecified
icon_box	String	Unspecified
title_font	Font	WM specific
icon_font	Font	WM specific
border_width	Int	WM specific
title_height	Int	WM specific
non_title_width	Int	WM specific
icon_name_width	Int	WM specific
iconify_width	Int	WM specific
iconify_height	Int	WM specific
icon_state	Bool	none

The `iconify_pixmap` is the icon pixmap to use. The `icon_box_x` is the icon x position in the icon box. The `icon_box_y` is the icon y position in the icon box. The `tiled` field indicates whether a tiled window manager is supported. The `sticky` field indicates whether sticky input focus is supported. The `no_iconify` field indicates whether it is to have an iconify button. The `no_lower` field indicates whether it is to have a lower button. The `no_resize` field indicates whether it is to have a resize button. The `icon_box` is the name to use for the icon box. The `title_font` is read-only and indicates the title font that is to be used. The `icon_font` is read-only and indicates the icon font that is to be used. The `border_width` is read-only and indicates the border width of the window manager. The `title_height` is read-only and indicates the height of the title bar. The `non_title_width` is read-only and indicates the width of the title bar less the title area. The `icon_name_width` is read-only and indicates the width that the window manager gives to the icon name. The `iconify_width` is read-only and indicates the width of the icon button. The `iconify_height` is read-only and indicates the height of the icon button. The `icon_state` is read-only and indicates whether it can be iconified.

Pop-Up Widgets 5

Pop-up widgets are used to create windows that are outside of the window hierarchy defined by the widget tree. Each pop-up child has a window that is a descendant of the root window so that the pop-up window is not clipped by the pop-up widget's parent window. Therefore, pop-ups are created and attached differently to their widget parent than from normal widget children.

A parent of a pop-up widget does not actively manage its pop-up children; in fact, it usually never notices them or operates upon them. The `popup_list` field in the `CorePart` structure contains the list of its pop-up children. This pop-up list exists mainly to provide the proper place in the widget hierarchy for the pop-up to get resources and to provide a place for `XtDestroyWidget` to look for all extant children.

A Composite widget can have both normal and pop-up children. A pop-up can be popped up from almost anywhere, not just by its parent. A child always refers to a normal, geometry-managed child on the children list, and a pop-up child always refers to a child on the pop-up list.

5.1 Pop-Up Widget Types

There are three kinds of pop-up widgets:

- **Modeless pop-ups**
A modeless pop-up (for example, a modeless dialog box) is usually visible to the window manager and looks like any other application from the user's point of view. (The application itself is a special form of a modeless pop-up.)
- **Modal pop-ups**
A modal pop-up (for example, a modal dialog box) may or may not be visible to the window manager and, except for events that occur in the dialog box, disables user-event processing by the application.
- **Spring-loaded pop-ups**
A spring-loaded pop-up (for example, a menu) is not visible to the window manager and, except for events that occur in the menu, disables user-event processing by all applications.

Modal pop-ups and spring-loaded pop-ups are very similar and should be coded as if they are the same. In fact, the same widget (for example, a `ButtonBox` or `Menu`) can be used both as a modal pop-up and as a spring-loaded pop-up within the same application. The main difference is that spring-loaded pop-ups are brought up with the pointer and, because of the grab that the pointer button causes, require different processing by the Intrinsics. Further, button up takes down a spring-loaded pop-up no matter where the button up occurs.

Any kind of pop-up, in turn, can pop up other widgets. Modal and spring-loaded pop-ups can constrain user events to the most recent such pop-up or to any of the modal/spring-loaded pop-ups currently mapped.

Regardless of their type, all pop-up widget classes are responsible for communicating with the X window manager and, therefore, are subclasses of `Shell`.

5.2 Creating a Pop-Up Shell

For a widget to pop up, it must be the child of a pop-up widget shell. A pop-up shell is never allowed more than one child, referred to as the pop-up child. Both the shell and child taken together are referred to as the pop-up. When you need to use a pop-up, you always should specify the pop-up shell, not the pop-up child.

To create a pop-up shell, use `XtCreatePopupShell`.

```
Widget XtCreatePopupShell(name, widget_class, parent,  
                          args, num_args)
```

```
String name;  
WidgetClass widget_class;  
Widget parent;  
ArgList args;  
Cardinal num_args;
```

name Specifies the text name for the created shell widget.

widget_class Specifies the widget class pointer for the created shell widget.

parent Specifies the parent widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in the argument list.

The `XtCreatePopupShell` function ensures that the specified class is a subclass of `Shell` and, rather than using `insert_child` to attach the widget to the parent's children list, attaches the shell to the parent's pop-ups list directly.

A spring-loaded pop-up invoked from a translation table already must exist at the time that the translation is invoked, so the translation manager can find the shell by name. Pop-ups invoked in other ways can be created “on-the-fly” when the pop-up actually is needed. This delayed creation of the shell is particularly useful when you pop up an unspecified number of pop-ups. You can look to see if an appropriate unused shell (that is, not currently popped up) exists and create a new shell if needed.

5.3 Creating Pop-Up Children

Once a pop-up shell is created, the single child of the pop-up shell can be created in one of two ways:

- Static
- Dynamic

At startup, an application can create the child of the pop-up shell, which is appropriate for pop-up children that are composed of a fixed set of widgets. The application can change the state of the subparts of the pop-up child as the application state changes. For example, if an application creates a static menu, it can call `XtSetSensitive` (or, in general, `XtSetValues`) on any of the buttons that make up the menu. Creating the pop-up child early means that pop-up time is minimized, especially if the application calls `XtRealizeWidget` on the pop-up shell at startup. When the menu is needed, all the widgets that make up the menu already exist and need only be mapped. The menu should pop up as quickly as the X server can respond.

Alternatively, an application can postpone the creation of the child until it is needed, which minimizes application startup time and allows the pop-up child to reconfigure itself each time it is popped up. In this case, the pop-up child creation routine should poll the application to find out if it should change the sensitivity of any of its subparts.

Pop-up child creation does not map the pop-up, even if you create the child and call `XtRealizeWidget` on the pop-up shell.

All shells have pop-up and pop-down callbacks, which provide the opportunity either to make last-minute changes to a pop-up child before it is popped up or to change it after it is popped down. Note that excessive use of pop-up callbacks can make popping up occur more slowly.

5.4 Mapping a Pop-Up Widget

Pop-ups can be popped up through several mechanisms:

- A call to `XtPopup`

- One of the supplied callback procedures (for example, `XtCallbackNone`, `XtCallbackNonexclusive`, or `XtCallbackExclusive`)
- The standard translation action `MenuPopup`

Some of these routines take an argument of type `XtGrabKind`, which is defined as:

```
typedef enum {XtGrabNone, XtGrabNonexclusive, XtGrabExclusive} XtGrabKind;
```

To map a pop-up from within an application, use `XtPopup`.

```
void XtPopup(popup_shell, grab_kind)
    Widget popup_shell;
    XtGrabKind grab_kind;
```

popup_shell Specifies the widget shell.

grab_kind Specifies the way in which user events should be constrained.

The `XtPopup` function performs the following:

- Calls `XtCheckSubclass` to ensure `popup_shell` is a subclass of `Shell`.
- Generates an error if the shell's `popped_up` field is already `True`.
- Calls the callback procedures on the shell's `popup_callback` list.
- Sets the shell `popped_up` field to `True`, the shell `spring_loaded` field to `False`, and the shell `grab_kind` field from `grab_kind`.
- If the shell's `create_popup_child` field is non-`NULL`, `XtPopup` calls it with `popup_shell` as the parameter.
- If `grab_kind` is either `XtGrabNonexclusive` or `XtGrabExclusive`, it calls:

```
XtAddGrab(popup_shell, (grab_kind == XtGrabExclusive), False)
```

- Calls `XtRealizeWidget` with `popup_shell` specified.
- Calls `XMapWindow` with `popup_shell` specified.

To map a pop-up from a given widget's callback list, you also can use the `XtCallbackNone`, `XtCallbackNonexclusive`, or `XtCallbackExclusive` convenience routines.

```
void XtCallbackNone(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    caddr_t call_data;
```

w Specifies the widget.
client_data Specifies the pop-up shell.
call_data Specifies the callback data, which is not used by this procedure.

```
void XtCallbackNonexclusive(w, client_data, call_data)  
    Widget w;  
    caddr_t client_data;  
    caddr_t call_data;
```

w Specifies the widget.
client_data Specifies the pop-up shell.
call_data Specifies the callback data, which is not used by this procedure.

```
void XtCallbackExclusive(w, client_data, call_data)  
    Widget w;  
    caddr_t client_data;  
    caddr_t call_data;
```

w Specifies the widget.
client_data Specifies the pop-up shell.
call_data Specifies the callback data, which is not used by this procedure.

The `XtCallbackNone`, `XtCallbackNonexclusive`, and `XtCallbackExclusive` functions call `XtPopup` with the shell specified by the client data argument and `grab_kind` set as the name specifies. `XtCallbackNone`, `XtCallbackNonexclusive`, and `XtCallbackExclusive` specify `XtGrabNone`, `XtGrabNonexclusive`, and `XtGrabExclusive`, respectively. Each function then sets the widget that executed the callback list to be insensitive by using `XtSetSensitive`. Using these functions in callbacks is not required. In particular, an application must provide customized code for callbacks that create pop-up shells dynamically or that must do more than desensitizing the button.

To pop up a menu when a pointer button is pressed or when the pointer is moved into some window, use `MenuPopup`. From a translation writer's point of view, the definition for this translation action is:

```
void MenuPopup(shell_name)  
    String shell_name;
```

shell_name Specifies the name of the widget shell to pop up.

MenuPopup is known to the translation manager, which must perform special actions for spring-loaded pop-ups. Calls to MenuPopup in a translation specification are mapped into calls to a nonexported action procedure, and the translation manager fills in parameters based on the event specified on the left-hand side of a translation.

If MenuPopup is invoked on ButtonPress (possibly with modifiers), the translation manager pops up the shell with grab_kind set to XtGrabExclusive and spring_loaded set to True. If MenuPopup is invoked on EnterWindow (possibly with modifiers), the translation manager pops up the shell with grab_kind set to XtGrabNonexclusive and spring_loaded set to False. Otherwise, the translation manager generates an error. When the widget is popped up, the following actions occur:

- Calls XtCheckSubclass to ensure popup_shell is a subclass of Shell.
- Generates an error if the shell's popped_up field is already True.
- Calls the callback procedures on the shell's popup_callback list.
- Sets the shell popped_up field to True and the shell grab_kind and spring_loaded fields appropriately.
- If the shell's create_popup_child field is non-NULL, it is called with popup_shell as the parameter.
- Calls:

```
XtAddGrab(popup_shell, (grab_kind == XtGrabExclusive), spring_loaded)
```

- Calls XtRealizeWidget with popup_shell specified.
- Calls XMapWindow with popup_shell specified.

(Note that these actions are the same as those for XtPopup.) MenuPopup tries to find the shell by searching the widget tree starting at the parent of the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that parent, it pops up the shell with the appropriate parameters. Otherwise, it moves up the parent chain as needed. If MenuPopup gets to the application widget and cannot find a matching shell, it generates an error.

5.5 Unmapping a Pop-Up Widget

Pop-ups can be popped down through several mechanisms:

- A call to XtPopdown

- The supplied callback procedure `XtCallbackPopdown`
- The standard translation action `MenuPopdown`

To unmap a pop-up from within an application, use `XtPopdown`.

```
void XtPopdown(popup_shell)
    Widget popup_shell;
```

popup_shell Specifies the widget shell to pop down.

The `XtPopdown` function performs the following:

- Calls `XtCheckSubclass` to ensure `popup_shell` is a subclass of `Shell`.
- Checks that `popup_shell` is currently popped_up; otherwise, it generates an error.
- Unmaps `popup_shell`'s window.
- If `popup_shell`'s `grab_kind` is either `XtGrabNonexclusive` or `XtGrabExclusive`, it calls `XtRemoveGrab`.
- Sets pop-up shell's `popped_up` field to `False`.
- Calls the callback procedures on the shell's `popupdown_callback` list.

To pop down pop-up that have been popped up with one of the callback routines (`XtCallbackNone`, `XtCallbackNonexclusive`, `XtCallbackExclusive`), use the callback `XtCallbackPopdown`.

```
void XtCallbackPopdown(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    caddr_t call_data;
```

w Specifies the widget.

client_data Specifies a pointer to the `XtPopdownID` structure.

call_data Specifies the callback data, which is not used by this procedure.

The `XtCallbackPopdown` function casts the client data parameter to an `XtPopdownID` pointer:

```
typedef struct {
    Widget shell_widget;
    Widget enable_widget;
} XtPopdownIDRec, *XtPopdownID;
```

The `shell_widget` is the pop-up shell to pop down, and the `enable_widget` is the widget that was used to pop it up.

XtCallbackPopdown calls XtPopdown with the specified `shell_widget` and then calls XtSetSensitive to resensitize the `enable_widget`.

To pop down a spring-loaded menu when a pointer button is released or when the pointer is moved into some window, use MenuPopdown. From a translation writer's point of view, the definition for this translation action is:

```
void MenuPopdown(shell_name)
    String shell_name;
```

shell_name Specifies the name of the widget shell to pop down.

If a shell name is not given, MenuPopdown calls XtPopdown with the widget for which the translation is specified. If a `shell_name` is specified in the translation table, MenuPopdown tries to find the shell by looking up the widget tree starting at the parent of the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that parent, it pops down the shell; otherwise, it moves up the parent chain as needed. If MenuPopdown gets to the application top-level shell widget and cannot find a matching shell, it generates an error.

Geometry Management 6

A widget does not directly control its size and location; rather, its parent is responsible for controlling its size and location. Although the position of children is usually left up to their parent, the widgets themselves often have the best idea of their optimal sizes and, possibly, preferred locations. To resolve physical layout conflicts between sibling widgets and between a widget and its parent, the Intrinsics provide the geometry management mechanism. Almost all Composite widgets have a geometry manager (`geometry_manager` field in the widget class record) that is responsible for the size, position, and stacking order of the widget's children. The only exception are fixed boxes, which create their children themselves and can ensure that their children will never make a geometry request.

6.1 Initiating Geometry Changes

Parents, children, and clients all initiate geometry changes differently. Because a parent has absolute control of its children's geometry, it changes the geometry directly by calling `XtMoveWidget`, `XtResizeWidget`, or `XtConfigureWidget`. A child must ask its parent for a geometry change by calling `XtMakeGeometryRequest` or `XtMakeResizeRequest` to convey its wishes to its parent. An application or other client code initiates a geometry change by calling `XtSetValues` on the appropriate geometry fields, thereby giving the widget the opportunity to modify or reject the client request before it gets propagated to the parent and the opportunity to respond appropriately to the parent's reply.

When a widget that needs to change its size, position, border width, or stacking depth asks its parent's geometry manager to make the desired changes, the geometry manager can do one of the following:

- Allow the request
- Disallow the request
- Suggest a compromise

When the geometry manager is asked to change the geometry of a child, the geometry manager may also rearrange and resize any or all of the other children that it controls. The geometry manager can move children

around freely using `XtMoveWidget`. When it resizes a child (that is, changes width, height, or `border_width`) other than the one making the request, it should do so by calling `XtResizeWidget`. It can simultaneously move and resize a child with a single call to `XtConfigureWidget`.

Often, geometry managers find that they can satisfy a request only if they can reconfigure a widget that they are not in control of (in particular, when the `Composite` widget wants to change its own size). In this case, the geometry manager makes a request to its parent's geometry manager. Geometry requests can cascade this way to arbitrary depth.

Because such cascaded arbitration of widget geometry can involve extended negotiation, windows are not actually allocated to widgets at application startup until all widgets are satisfied with their geometry. For further information, see Sections 2.4 and 2.5.

Note

1. The Intrinsic treatment of stacking requests is deficient in several areas. Stacking requests for unrealized widgets are granted but will have no effect. In addition, there is no way to do an `XtSetValues` that will generate a stacking geometry request.
2. After a successful geometry request (one that returned `XtGeometryYes`), a widget does not know whether or not its resize procedure has been called. Widgets should have resize procedures that can be called more than once without ill effects.

6.2 General Geometry Manager Requests

To make a general geometry manager request from a widget, use `XtMakeGeometryRequest`.

```
XtGeometryResult XtMakeGeometryRequest(w, request,  
                                         reply_return)
```

```
Widget w;  
XtWidgetGeometry *request;  
XtWidgetGeometry *reply_return;
```

- | | |
|---------------------|---|
| <i>w</i> | Specifies the widget that is making the request. |
| <i>request</i> | Specifies the desired widget geometry (size, position, border width, and stacking order). |
| <i>reply_return</i> | Returns the allowed widget size or may be NULL if the requesting widget is not interested in handling <code>XtGeometryAlmost</code> . |

Depending on the condition, `XtMakeGeometryRequest` performs the following:

- If the widget is unmanaged or the widget's parent is not realized, it makes the changes and returns `XtGeometryYes`.
- If the parent is not a subclass of `compositeWidgetClass` or the parent's `geometry_manager` is `NULL`, it issues an error.
- If the widget's `being_destroyed` field is `True`, it returns `XtGeometryNo`.
- If the widget `x`, `y`, `width`, `height` and `border_width` fields are all equal to the requested values, it returns `XtGeometryYes`; otherwise, it calls the parent's `geometry_manager` procedure with the given parameters.
- If the parent's geometry manager returns `XtGeometryYes` and if `XtCWQueryOnly` is not set in the `request_mode` and if the widget is realized, `XtMakeGeometryRequest` calls the `XConfigureWindow` Xlib function to reconfigure the widget's window (set its size, location, and stacking order as appropriate).
- If the geometry manager returns `XtGeometryDone`, the change has been approved and actually has been done. In this case, `XtMakeGeometryRequest` does no configuring and returns `XtGeometryYes`. `XtMakeGeometryRequest` never returns `XtGeometryDone`.

Otherwise, `XtMakeGeometryRequest` returns the resulting value from the parent's geometry manager.

Children of primitive widgets are always unmanaged; thus, `XtMakeGeometryRequest` always returns `XtGeometryYes` when called by a child of a primitive widget.

The return codes from geometry managers are:

```
typedef enum _XtGeometryResult {
    XtGeometryYes,
    XtGeometryNo,
    XtGeometryAlmost,
    XtGeometryDone
} XtGeometryResult;
```

The `XtWidgetGeometry` structure is quite similar but not identical to the corresponding Xlib structure:

```
typedef unsigned long XtGeometryMask;

typedef struct {
    XtGeometryMask request_mode;
    Position x, y;
    Dimension width, height;
    Dimension border_width;
```

```
    Widget sibling;
    int stack_mode;
} XtWidgetGeometry;
```

The `request_mode` definitions are from `<X11/X.h>`:

```
#define CWX (1<<0)
#define CWY (1<<1)
#define CWWidth (1<<2)
#define CWHeight (1<<3)
#define CWBorderWidth (1<<4)
#define CWSibling (1<<5)
#define CWStackMode (1<<6)
```

The Intrinsic also support the following value:

```
#define XtCWQueryOnly (1<<7)
```

`XtCWQueryOnly` indicates that the corresponding geometry request is only a query as to what would happen if this geometry request were made and that no widgets should actually be changed.

`XtMakeGeometryRequest`, like the `XConfigureWindow` Xlib function, uses `request_mode` to determine which fields in the `XtWidgetGeometry` structure you want to specify.

The `stack_mode` definitions are from `<X11/X.h>`:

```
#define Above 0
#define Below 1
#define TopIf 2
#define BottomIf 3
#define Opposite 4
```

The Intrinsic also support the following value:

```
#define XtSMDontChange 5
```

For definition and behavior of `Above`, `Below`, `TopIf`, `BottomIf`, and `Opposite`, see the *Guide to the Xlib Library*. `XtSMDontChange` indicates that the widget wants its current stacking order preserved.

6.3 Resize Requests

To make a simple resize request from a widget, you can use `XtMakeResizeRequest` as an alternative to `XtMakeGeometryRequest`.

```
XtGeometryResult XtMakeResizeRequest(w, width, height,  
                                     width_return,  
                                     height_return)  
  
Widget w;  
Dimension width, height;  
Dimension *width_return, *height_return
```

w Specifies the widget.

width

height Specify the desired widget width and height.

width_return

height_return Return the allowed widget width and height.

The `XtMakeResizeRequest` function, a simple interface to `XtMakeGeometryRequest`, creates a `XtWidgetGeometry` structure and specifies that width and height should change. The geometry manager is free to modify any of the other window attributes (position or stacking order) to satisfy the resize request. If the return value is `XtGeometryAlmost`, `width_return` and `height_return` contain a compromise width and height. If these are acceptable, the widget should immediately make an `XtMakeResizeRequest` and request that the compromise width and height be applied. If the widget is not interested in `XtGeometryAlmost` replies, it can pass `NULL` for `width_return` and `height_return`.

6.4 Potential Geometry Changes

Sometimes a geometry manager cannot respond to a geometry request from a child without first making a geometry request to the widget's own parent (the requestor's grandparent). If the request to the grandparent would allow the parent to satisfy the original request, the geometry manager can make the intermediate geometry request as if it were the originator. On the other hand, if the geometry manager already has determined that the original request cannot be completely satisfied (for example, if it always denies position changes), it needs to tell the grandparent to respond to the intermediate request without actually changing the geometry because it does not know if the child will accept the compromise. To accomplish this, the geometry manager uses `XtCWQueryOnly` in the intermediate request.

When `XtCWQueryOnly` is used, the geometry manager needs to cache enough information to exactly reconstruct the intermediate request. If the grandparent's response to the intermediate query was `XtGeometryAlmost`, the geometry manager needs to cache the entire reply geometry in the event the child accepts the parent's compromise.

If the grandparent's response was `XtGeometryAlmost`, it may also be necessary to cache the entire reply geometry from the grandparent when `XtCWQueryOnly` is not used. If the geometry manager is still able to satisfy the original request, it may immediately accept the grandparent's compromise and then act on the child's request. If the grandparent's compromise geometry is insufficient to allow the child's request and if the geometry manager is willing to offer a different compromise to the child, the grandparent's compromise should not be accepted until the child has accepted the new compromise.

Note that a compromise geometry returned with `XtGeometryAlmost` is guaranteed only for the next call to the same widget; therefore, a cache of size one is sufficient.

6.5 Child Geometry Management: the `geometry_manager` Procedure

The `geometry_manager` procedure pointer in a composite widget class is of type `XtGeometryHandler`:

```
typedef XtGeometryResult (*XtGeometryHandler)(Widget,
                                              XtWidgetGeometry *,
                                              XtWidgetGeometry *);

Widget w;
XtWidgetGeometry *request;
XtWidgetGeometry *geometry_return;
```

A class can inherit its superclass's geometry manager during class initialization.

A bit set to zero in the request's mask field means that the child widget does not care about the value of the corresponding field. Then, the geometry manager can change it as it wishes. A bit set to 1 means that the child wants that geometry element changed to the value in the corresponding field.

If the geometry manager can satisfy all changes requested and if `XtCWQueryOnly` is not specified, it updates the widget's `x`, `y`, `width`, `height`, and `border_width` values appropriately. Then, it returns `XtGeometryYes`, and the value of the `geometry_return` argument is undefined. The widget's window is moved and resized automatically by `XtMakeGeometryRequest`.

Homogeneous composite widgets often find it convenient to treat the widget making the request the same as any other widget, possibly reconfiguring it as part of its layout process, unless `XtCWQueryOnly` is specified. If it does this, it should return `XtGeometryDone` to inform `XtMakeGeometryRequest` that it does not need to do the configuration itself.

Although `XtMakeGeometryRequest` resizes the widget's window (if the geometry manager returns `XtGeometryYes`), it does not call the widget class's resize procedure. The requesting widget must perform whatever resizing calculations are needed explicitly.

If the geometry manager chooses to disallow the request, the widget cannot change its geometry. The value of the `geometry_return` parameter is undefined, and the geometry manager returns `XtGeometryNo`.

Sometimes the geometry manager cannot satisfy the request exactly, but it may be able to satisfy a similar request. That is, it could satisfy only a subset of the requests (for example, size but not position) or a lesser request (for example, it cannot make the child as big as the request but it can make the child bigger than its current size). In such cases, the geometry manager fills in `geometry_return` with the actual changes it is willing to make, including an appropriate mask, and returns `XtGeometryAlmost`. If a bit in `geometry_return->request_mode` is zero, the geometry manager does not change the corresponding value if the `geometry_return` is used immediately in a new request. If a bit is one, the geometry manager does change that element to the corresponding value in `geometry_return`. More bits may be set in `geometry_return` than in the original request if the geometry manager intends to change other fields should the child accept the compromise.

When `XtGeometryAlmost` is returned, the widget must decide if the compromise suggested in `geometry_return` is acceptable. If it is, the widget must not change its geometry directly; rather, it must make another call to `XtMakeGeometryRequest`.

If the next geometry request from this child uses the `geometry_return` box filled in by an `XtGeometryAlmost` return and if there have been no intervening geometry requests on either its parent or any of its other children, the geometry manager must grant the request, if possible. That is, if the child asks immediately with the returned geometry, it should get an answer of `XtGeometryYes`. However, the user's window manager may affect the final outcome.

To return an `XtGeometryYes`, the geometry manager frequently rearranges the position of other managed children by calling `XtMoveWidget`. However, a few geometry managers may sometimes change the size of other managed children by calling `XtResizeWidget` or `XtConfigureWidget`. If `XtCWQueryOnly` is specified, the geometry manager must return how it would react to this geometry request without actually moving or resizing any widgets.

Geometry managers must not assume that the request and `geometry_return` arguments point to independent storage. The caller is permitted to use the same field for both, and the geometry manager must allocate its own temporary storage, if necessary.

6.6 Widget Placement and Sizing

To move a sibling widget of the child making the geometry request, use `XtMoveWidget`.

```
void XtMoveWidget(w, x, y)
    Widget w;
    Position x;
    Position y;
```

w Specifies the widget.

x

y Specify the new widget *x* and *y* coordinates.

The `XtMoveWidget` function returns immediately if the specified geometry fields are the same as the old values. Otherwise, `XtMoveWidget` writes the new *x* and *y* values into the widget and, if the widget is realized, issues an Xlib `XMoveWindow` call on the widget's window.

To resize a sibling widget of the child making the geometry request, use `XtResizeWidget`.

```
void XtResizeWidget(w, width, height, border_width)
    Widget w;
    Dimension width;
    Dimension height;
    Dimension border_width;
```

w Specifies the widget.

width

height

border_width Specify the new widget size.

The `XtResizeWidget` function returns immediately if the specified geometry fields are the same as the old values. Otherwise, `XtResizeWidget` writes the new *width*, *height*, and *border_width* values into the widget and, if the widget is realized, issues an `XConfigureWindow` call on the widget's window.

If the new *width* or *height* are different from the old values, `XtResizeWidget` calls the widget's resize procedure to notify it of the size change.

To move and resize the sibling widget of the child making the geometry request, use `XtConfigureWidget`.

```
void XtConfigureWidget(w, x, y, width, height, border_width)
    Widget w;
    Position x;
    Position y;
    Dimension width;
    Dimension height;
    Dimension border_width;
```

w Specifies the widget.

x

y Specify the new widget x and y coordinates.

width

height

border_width Specify the new widget size.

The XtConfigureWidget function returns immediately if the specified geometry fields are the same as the old values. Otherwise, XtConfigureWidget writes the new x, y, width, height, and border_width values into the widget and, if the widget is realized, makes an Xlib XConfigureWindow call on the widget's window.

If either the new width or height is different from its old value, XtConfigureWidget calls the widget's resize procedure to notify it of the size change; otherwise, it simply returns.

To resize a child widget that already has the new values of its width, height, and border width fields, use XtResizeWindow.

```
void XtResizeWindow(w)
    Widget w;
```

w Specifies the widget.

The XtResizeWindow function calls the XConfigureWindow Xlib function to make the window of the specified widget match its width, height, and border width. This request is done unconditionally because there is no way to tell if these values match the current values. Note that the widget's resize procedure is not called.

There are very few times to use XtResizeWindow; instead, you should use XtResizeWidget.

6.7 Preferred Geometry

Some parents may be willing to adjust their layouts to accommodate the preferred geometries of their children. They can use XtQueryGeometry

to obtain the preferred geometry and, as they see fit, can use or ignore any portion of the response.

To query a child widget's preferred geometry, use `XtQueryGeometry`.

```
XtGeometryResult XtQueryGeometry(w, intended, preferred_return  
    Widget w;  
    XtWidgetGeometry *intended, *preferred_return;
```

w Specifies the widget.

intended Specifies any changes the parent plans to make to the child's geometry or NULL.

preferred_return

Returns the child widget's preferred geometry.

To discover a child's preferred geometry, the child's parent sets any changes that it intends to make to the child's geometry in the corresponding fields of the *intended* structure, sets the corresponding bits in *intended.request_mode*, and calls `XtQueryGeometry`.

`XtQueryGeometry` clears all bits in the *preferred_return->request_mode* and checks the *query_geometry* field of the specified widget's class record. If *query_geometry* is not NULL, `XtQueryGeometry` calls the *query_geometry* procedure and passes as arguments the specified widget, *intended*, and *preferred_return* structures. If the *intended* argument is NULL, `XtQueryGeometry` replaces it with a pointer to an `XtWidgetGeometry` structure with *request_mode*=0 before calling *query_geometry*.

The *query_geometry* procedure pointer is of type `XtGeometryHandler`.

```
typedef XtGeometryResult (*XtGeometryHandler)(Widget,  
                                              XtWidgetGeometry *,  
                                              XtWidgetGeometry *);  
  
Widget w;  
XtWidgetGeometry *request;  
XtWidgetGeometry *geometry_return;
```

The *query_geometry* procedure is expected to examine the bits set in *request->request_mode*, evaluate the preferred geometry of the widget, and store the result in *geometry_return* (setting the bits in *geometry_return->request_mode* corresponding to those geometry fields that it cares about). If the proposed geometry change is acceptable without modification, the *query_geometry* procedure should return `XtGeometryYes`. If at least one field in *geometry_return* is different from the corresponding field in *request* or if a bit was set in *geometry_return* that was not set in *request*, the *query_geometry* procedure should return `XtGeometryAlmost`. If the preferred

geometry is identical to the current geometry, the `query_geometry` procedure should return `XtGeometryNo`.

After calling the `query_geometry` procedure or if the `query_geometry` field is `NULL`, `XtQueryGeometry` examines all the unset bits in `geometry_return->request_mode` and sets the corresponding fields in `geometry_return` to the current values from the widget instance. If `CWStackMode` is not set, the `stack_mode` field is set to `XtSMDontChange`. `XtQueryGeometry` returns the value returned by the `query_geometry` procedure or `XtGeometryYes` if the `query_geometry` field is `NULL`.

Therefore, the caller can interpret a return of `XtGeometryYes` as not needing to evaluate the contents of `reply` and, more importantly, not needing to modify its layout plans. A return of `XtGeometryAlmost` means either that both the parent and the child expressed interest in at least one common field and the child's preference does not match the parent's intentions or that the child expressed interest in a field that the parent might need to consider. A return value of `XtGeometryNo` means that both the parent and the child expressed interest in a field and that the child suggests that the field's current value is its preferred value. In addition, whether or not the caller ignores the return value or the reply mask, it is guaranteed that the reply structure contains complete geometry information for the child.

Parents are expected to call `XtQueryGeometry` in their layout routine and wherever other information is significant after `change_managed` has been called. The `changed_managed` procedure may assume that the child's current geometry is its preferred geometry. Thus, the child is still responsible for storing values into its own geometry during its initialize procedure.

6.8 Size Change Management: the `resize` Procedure

A child can be resized by its parent at any time. Widgets usually need to know when they have changed size so that they can lay out their displayed data again to match the new size. When a parent resizes a child, it calls `XtResizeWidget`, which updates the geometry fields in the widget, configures the window if the widget is realized, and calls the child's `resize` procedure to notify the child. The `resize` procedure pointer is of type `XtWidgetProc`.

If a class need not recalculate anything when a widget is resized, it can specify `NULL` for the `resize` field in its class record. This is an unusual case and should occur only for widgets with very trivial display semantics. The `resize` procedure takes a widget as its only argument. The `x`, `y`, `width`, `height` and `border_width` fields of the widget contain the new values. The `resize` procedure should recalculate the layout of internal data as needed. (For example, a centered `Label` in a window that changes size

should recalculate the starting position of the text.) The widget must obey resize as a command and must not treat it as a request. A widget must not issue an `XtMakeGeometryRequest` or `XtMakeResizeRequest` call from its resize procedure.

Event Management 7

While X allows the reading and processing of events anywhere in an application, widgets in the XUI Toolkit neither directly read events nor grab the server or pointer. Widgets register procedures that are to be called when an event or class of events occurs in that widget.

A typical application consists of startup code followed by an event loop that reads events and dispatches them by calling the procedures that widgets have registered. The default event loop provided by the Intrinsics is `XtAppMainLoop`.

The event manager is a collection of functions to perform the following tasks:

- Add or remove event sources other than X server events (in particular, timer interrupts and file input).
- Query the status of event sources.
- Add or remove procedures to be called when an event occurs for a particular widget.
- Enable and disable the dispatching of user-initiated events (keyboard and pointer events) for a particular widget.
- Constrain the dispatching of events to a cascade of pop-up widgets.
- Call the appropriate set of procedures currently registered when an event is read.

Most widgets do not need to call any of the event handler functions explicitly. The normal interface to X events is through the higher-level translation manager, which maps sequences of X events (with modifiers) into procedure calls. Applications rarely use any of the event manager routines besides `XtAppMainLoop`.

7.1 Adding and Deleting Additional Event Sources

While most applications are driven only by X events, some applications need to incorporate other sources of input into the XUI Toolkit event handling mechanism. The event manager provides routines to integrate notification of timer events and file data pending into this mechanism.

The next section describes functions that provide input gathering from files. The application registers the files with the Intrinsics read routine. When input is pending on one of the files, the registered callback procedures are invoked.

7.1.1 Adding and Removing Input Sources

To register a new file as an input source for a given application, use `XtAppAddInput`.

```
XtInputId XtAppAddInput(app_context, source, condition, proc,  
                        client_data)
```

```
    XtAppContext app_context;  
    int source;  
    caddr_t condition;  
    XtInputCallbackProc proc;  
    caddr_t client_data;
```

- app_context* Specifies the application context that identifies the application.
- source* Specifies the source file descriptor on a UNIX-based system or other operating system dependent device specification.
- condition* Specifies the mask that indicates a read, write, or exception condition or some operating system dependent condition.
- proc* Specifies the procedure that is to be called when input is available.
- client_data* Specifies the argument that is to be passed to the specified procedure when input is available.

The `XtAppAddInput` function registers with the Intrinsics read routine a new source of events, which is usually file input but can also be file output. Note that file should be loosely interpreted to mean any sink or source of data. `XtAppAddInput` also specifies the conditions under which the source can generate events. When input is pending on this source, the callback procedure is called.

The legal values for the `condition` argument are operating-system dependent. On a UNIX-based system, the `condition` is some union of `XtInputReadMask`, `XtInputWriteMask`, and `XtInputExceptMask`.

Callback procedure pointers that are used when there are file events are of type `XtInputCallbackProc`:

```
typedef void (*XtInputCallbackProc)(caddr_t, int *,  
                                    XtInputId *);  
  
    caddr_t client_data;  
    int *source;  
    XtInputId *id;
```

client_data Specifies the client data that was registered for this procedure in `XtAppAddInput`.

source Specifies the source file descriptor generating the event.

id Specifies the ID returned from the corresponding `XtAppAddInput` call.

To discontinue a source of input, use `XtRemoveInput`.

```
void XtRemoveInput(id)
    XtInputId id;
```

id Specifies the ID returned from the corresponding `XtAppAddInput` call.

The `XtRemoveInput` function causes the Intrinsics read routine to stop watching for input from the input source.

7.1.2 Adding and Removing Timeouts

The timeout facility notifies the application or the widget through a callback procedure that a specified time interval has elapsed. Timeout values are uniquely identified by an interval ID.

To create a timeout value, use `XtAppAddTimeOut`.

```
XtIntervalId XtAppAddTimeOut(app_context, interval, proc,
                             client_data)
    XtAppContext app_context;
    unsigned long interval;
    XtTimerCallbackProc proc;
    caddr_t client_data;
```

app_context Specifies the application context for which the timer is to be set.

interval Specifies the time interval in milliseconds.

proc Specifies the procedure that is to be called when the time expires.

client_data Specifies the argument that is to be passed to the specified procedure when it is called.

The `XtAppAddTimeout` function creates a timeout and returns an identifier for it. The timeout value is set to `interval`. The callback procedure is called when the time interval elapses, and then the timeout is removed. Callback procedure pointer that are used when timeouts expire are of type `XtTimerCallbackProc`:

```
typedef void (*XtTimerCallbackProc)(caddr_t,  
                                     XtIntervalId *);  
  
    caddr_t client_data;  
    XtIntervalId *id;
```

client_data Specifies the client data that was registered for this procedure in `XtAppAddTimeout`.

id Specifies the ID returned from the corresponding `XtAppAddTimeout` call.

To clear a timeout value, use `XtRemoveTimeout`.

```
void XtRemoveTimeout(timer)  
    XtIntervalId timer;
```

timer Specifies the ID for the timeout request to be destroyed.

The `XtRemoveTimeout` function removes the timeout. Note that timeouts are automatically removed once they trigger.

7.2 Constraining Events to a Cascade of Widgets

Modal widgets are widgets that, except for the input directly to them, lock out user input to the application.

When a modal menu or modal dialog box is popped up using `XtPopup`, user events (keyboard and pointer events) that occur outside the modal widget should be delivered to the modal widget or ignored. In no case will user events be delivered to a widget outside the modal widget.

Menus can pop up submenus and dialog boxes can pop up further dialog boxes to create a pop-up cascade. In this case, user events may be delivered to one of several modal widgets in the cascade.

Display-related events should be delivered outside the modal cascade so that expose events and the like keep the application's display up to date. Any event that occurs within the cascade is delivered as usual. The user events that are delivered to the most recent spring-loaded shell in the cascade when they occur outside the cascade are called remap events and are `KeyPress`, `KeyRelease`, `ButtonPress`, and `ButtonRelease`. The user events that are ignored when they occur outside the cascade are `MotionNotify`, `EnterNotify`, and `LeaveNotify`. All other events are delivered normally.

XtPopup uses the XtAddGrab and XtRemoveGrab functions to constrain user events to a modal cascade and subsequently to remove a grab when the modal widget goes away. Usually you should have no need to call them explicitly.

To redirect user input to a modal widget, use XtAddGrab.

```
void XtAddGrab(w, exclusive, spring_loaded)
    Widget w;
    Boolean exclusive;
    Boolean spring_loaded;
```

w Specifies the widget to add to the modal cascade.

exclusive Specifies whether user events should be dispatched exclusively to this widget or also to previous widgets in the cascade.

spring_loaded Specifies whether this widget was popped up because the user pressed a pointer button.

The XtAddGrab function appends the widget (and associated parameters) to the modal cascade and checks that exclusive is True if spring_loaded is True. If these are not True, XtAddGrab generates an error.

The modal cascade is used by XtDispatchEvent when it tries to dispatch a user event. When at least one modal widget is in the widget cascade, XtDispatchEvent first determines if the event should be delivered. It starts at the most recent cascade entry and follows the cascade up to and including the most recent cascade entry added with the exclusive parameter True.

This subset of the modal cascade along with all descendants of these widgets comprise the active subset. User events that occur outside the widgets in this subset are ignored or remapped. Modal menus with submenus generally add a submenu widget to the cascade with exclusive False. Modal dialog boxes that need to restrict user input to the most deeply nested dialog box add a subdialog widget to the cascade with exclusive True. User events that occur within the active subset are delivered to the appropriate widget, which is usually a child or further descendant of the modal widget.

Regardless of where on the screen they occur, remap events are always delivered to the most recent widget in the active subset of the cascade that has spring_loaded True, if any such widget exists.

To remove the redirection of user input to a modal widget, use `XtRemoveGrab`.

```
void XtRemoveGrab(w)
    Widget w;
```

w Specifies the widget to remove from the modal cascade.

The `XtRemoveGrab` function removes widgets from the modal cascade starting at the most recent widget up to and including the specified widget. It issues an error if the specified widget is not on the modal cascade.

7.3 Focusing Events on a Child

To redirect keyboard input to a child of a `Composite` widget without calling `XSetInputFocus`, use `XtSetKeyboardFocus`.

```
XtSetKeyboardFocus(subtree, descendant)
    Widget subtree, descendant;
```

subtree Specifies the subtree of the hierarchy for which the keyboard focus is to be set.

descendant Specifies either the widget in the subtree structure which is to receive the keyboard event, or `None`. Note that it is not an error to specify `None` when no input focus was previously set.

If a future `KeyPress` or `KeyRelease` event occurs within the specified subtree, `XtSetKeyboardFocus` causes `XtDispatchEvent` to remap and send the event to the specified descendant widget.

When there is no modal cascade, keyboard events can occur within a widget `W` in one of three ways:

- `W` has the X input focus.
- `W` has the keyboard focus of one of its ancestors, and the event occurs within the ancestor or one of the ancestor's descendants.
- No ancestor of `W` has a descendant within the keyboard focus, and the pointer is within `W`.

When there is a modal cascade, a widget `W` receives keyboard events if an ancestor of `W` is in the active subset of the modal cascade and one or more of the previous conditions is `True`.

When `subtree` or one of its descendants acquires the X input focus or the pointer moves into the subtree such that keyboard events would now be delivered to `subtree`, a `FocusIn` event is generated for the descendant if `FocusNotify` events have been selected by the descendant. Similarly, when `W` loses the X input focus or the keyboard focus for one of its ancestors,

a `FocusOut` event is generated for descendant if `FocusNotify` events have been selected by the descendant.

The `accept_focus` procedure pointer is of type `XtAcceptFocusProc`:

```
typedef Boolean (*XtAcceptFocusProc)(Widget, Time);
Widget w;
Time *time;
```

w Specifies the widget.

time Specifies the X time of the event causing the accept focus.

Widgets that need the input focus can call `XSetInputFocus` explicitly. To allow outside agents to cause a widget to get the input focus, every widget exports an `accept_focus` procedure. The widget returns whether it actually took the focus or not, so that the parent can give the focus to another widget. Widgets that need to know when they lose the input focus must use the Xlib focus notification mechanism explicitly (typically by specifying translations for `FocusIn` and `FocusOut` events). Widgets that never want the input focus should set their `accept_focus` procedure pointer to `NULL`.

To call a widget's `accept_focus` procedure, use `XtCallAcceptFocus`.

```
Boolean XtCallAcceptFocus(w, time)
Widget w;
Time *time;
```

w Specifies the widget.

time Specifies the X time of the event that is causing the accept focus.

The `XtCallAcceptFocus` function calls the specified widget's `accept_focus` procedure, passing it the specified widget and time, and returns what the `accept_focus` procedure returns. If `accept_focus` is `NULL`, `XtCallAcceptFocus` returns `False`.

7.4 Querying Event Sources

The event manager provides several functions to examine and read events (including file and timer events) that are in the queue. The next three functions handle Intrinsic equivalents of the `XPending`, `XPeekEvent`, and `XNextEvent` Xlib calls.

To determine if there are any events on the input queue for a given application, use `XtAppPending`.

```
XtInputMask XtAppPending(app_context)
    XtAppContext app_context;
```

app_context Specifies the application context that identifies the application to check.

The `XtAppPending` function returns a nonzero value if there are events pending from the X server, timer pending, or other input sources pending. The value returned is a bit mask that is the OR of `XtIMXEvent`, `XtIMTimer`, and `XtIMAlternateInput` (see `XtAppProcessEvent`). If there are no events pending, `XtAppPending` flushes the output buffer and returns zero.

To return the value from the head of a given application's input queue without removing input from the queue, use `XtAppPeekEvent`.

```
Boolean XtAppPeekEvent(app_context, event_return)
    XtAppContext app_context;
    XEvent *event_return;
```

app_context Specifies the application context that identifies the application.

event_return Returns the event information to the specified event structure.

If there is an event in the queue, `XtAppPeekEvent` fills in the event and returns a nonzero value. If no X input is on the queue, `XtAppPeekEvent` flushes the output buffer and blocks until input is available (possibly calling some timeout callbacks in the process). If the input is an event, `XtAppPeekEvent` fills in the event and returns a nonzero value. Otherwise, the input is for an alternate input source, and `XtAppPeekEvent` returns zero.

To return the value from the head of a given application's input queue, use `XtAppNextEvent`.

```
void XtAppNextEvent(app_context, event_return)
    XtAppContext app_context;
    XEvent *event_return;
```

app_context Specifies the application context that identifies the application.

event_return Returns the event information to the specified event structure.

If no input is on the X input queue, `XtAppNextEvent` flushes the X output buffer and waits for an event while looking at the other input sources and timeout values and calling any callback procedures triggered by them. This wait time can be used for background processing (see Section 7.8).

7.5 Dispatching Events

The Intrinsics provide functions that dispatch events to widgets or other application code. Every client interested in X events on a widget uses `XtAddEventHandler` to register which events it is interested in and a procedure (event handler) that is to be called when the event happens in that window. The translation manager automatically registers event handlers for widgets that use translation tables (see Chapter 10).

Applications that need direct control of the processing of different types of input should use `XtAppProcessEvent`.

```
void XtAppProcessEvent(app_context, mask)
    XtAppContext app_context;
    XtInputMask mask;
```

app_context Specifies the application context that identifies the application for which to process input.

mask Specifies what types of events to process. The mask is the bitwise inclusive OR of any combination of `XtIMXEvent`, `XtIMTimer`, and `XtIMAlternateInput`. As a convenience, the XUI Toolkit defines the symbolic name `XtIMAll` to be the bitwise inclusive OR of all event types.

The `XtAppProcessEvent` function processes one timer, alternate input, or X event. If there is nothing of the appropriate type to process, `XtAppProcessEvent` blocks until there is. If there is more than one type of thing available to process, it is undefined which will get processed.

Usually, this procedure is not called by client applications (see `XtAppMainLoop`). `XtAppProcessEvent` processes timer events by calling any appropriate timer callbacks, alternate input by calling any appropriate alternate input callbacks, and X events by calling `XtDispatchEvent`.

When an X event is received, it is passed to `XtDispatchEvent`, which calls the appropriate event handlers and passes them the widget, the event, and client-specific data registered with each procedure. If there are no handlers for that event registered, the event is ignored and the dispatcher simply returns. The order in which the handlers are called is undefined.

```
Boolean XtDispatchEvent(event)
    XEvent *event;
```

event Specifies a pointer to the event structure that is to be dispatched to the appropriate event handler.

The XtDispatchEvent function sends those events to the event handler functions that have been previously registered with the dispatch routine. XtDispatchEvent returns True if it dispatched the event to some handler and False if it found no handler to dispatch the event to. The most common use of XtDispatchEvent is to dispatch events acquired with the XtAppNextEvent procedure. However, it also can be used to dispatch user-constructed events. XtDispatchEvent also is responsible for implementing the grab semantics for XtAddGrab.

7.6 The Application Input Loop

To process input from a given application, use XtAppMainLoop.

```
void XtAppMainLoop(app_context)
    XtAppContext app_context;
```

app_context Specifies the application context that identifies the application.

The XtAppMainLoop function first reads the next incoming X event by calling XtAppNextEvent and then it dispatches the event to the appropriate registered procedure by calling XtDispatchEvent. This constitutes the main loop of XUI Toolkit applications, and, as such, it does not return. Applications are expected to exit in response to some user action. There is nothing special about XtAppMainLoop; it is simply an infinite loop that calls XtAppNextEvent and then XtDispatchEvent.

Applications can provide their own version of this loop, which tests some global termination flag or tests that the number of top-level widgets is larger than zero before circling back to the call to XtAppNextEvent.

7.7 Setting and Checking the Sensitivity State of a Widget

Many widgets have a mode in which they assume a different appearance (for example, are greyed out or stippled), do not respond to user events, and become dormant.

When dormant, a widget is considered to be insensitive. If a widget is insensitive, the Event Manager does not dispatch any events to the widget with an event type of KeyPress, KeyRelease, ButtonPress, ButtonRelease, MotionNotify, EnterNotify, LeaveNotify, FocusIn, or FocusOut.

A widget can be insensitive because its sensitive field is `False` or because one of its parents is insensitive, and, thus, the widget's `ancestor_sensitive` field also is `False`. A widget can but does not need to distinguish these two cases visually.

To set the sensitivity state of a widget, use `XtSetSensitive`.

```
void XtSetSensitive(w, sensitive)
    Widget w;
    Boolean sensitive;
```

w Specifies the widget.

sensitive Specifies a Boolean value that indicates whether the widget should receive keyboard and pointer events.

The `XtSetSensitive` function first calls `XtSetValues` on the current widget with an argument list specifying that the sensitive field should change to the new value. It then recursively propagates the new value down the managed children tree by calling `XtSetValues` on each child to set the `ancestor_sensitive` to the new value if the new values for sensitive and the child's `ancestor_sensitive` are not the same.

`XtSetSensitive` calls `XtSetValues` to change sensitive and `ancestor_sensitive`. Therefore, when one of these changes, the widget's `set_values` procedure should take whatever display actions are needed (for example, greying out or stippling the widget).

`XtSetSensitive` maintains the invariant that if parent has either sensitive or `ancestor_sensitive` `False`, then all children have `ancestor_sensitive` `False`.

To check the current sensitivity state of a given widget (which is usually done by parents), use `XtIsSensitive`.

```
Boolean XtIsSensitive(w)
    Widget w;
```

w Specifies the widget.

The `XtIsSensitive` function returns `True` or `False` to indicate whether or not user input events are being dispatched. If both `core_sensitive` and `core_ancestor_sensitive` are `True`, `XtIsSensitive` returns `True`; otherwise, it returns `False`.

7.8 Adding Background Work Procedures

The Intrinsics have limited support for background processing. Because most applications spend most of their time waiting for input, you can register an idle-time work procedure that will be called when the toolkit would otherwise block in `XtAppNextEvent` or `XtAppProcessEvent`. Work

procedure pointers are of type `XtWorkProc`:

```
typedef Boolean (*XtWorkProc)(caddr_t);
    caddr_t client_data;
```

client_data Client data specified when the work proc was registered.

This procedure returns `True` if it is done, that is, the work procedure should be removed. Work procedures should be very judicious about how much they do. If they run for more than a small part of a second, response time is likely to suffer.

To register a work procedure for a given application, use `XtAppAddWorkProc`.

```
XtWorkProcId XtAppAddWorkProc(app_context, proc, client_data)
    XtAppContext app_context;
    XtWorkProc proc;
    caddr_t client_data;
```

app_context Specifies the application context that identifies the application.

proc Specifies the procedure that is to be called when the application is idle.

client_data Specifies the argument that is to be passed to the specified procedure when it is called.

The `XtAppAddWorkProc` function adds the specified work procedure for the application identified by *app_context*.

`XtWorkProcId` is an opaque unique identifier for this work procedure. Multiple work procedures can be registered, and the most recently added one is always the one that is called. However, if a work procedure adds another work procedure, the newly added one has lower priority than the current one.

To remove a work procedure, either return `True` from the procedure when it is called or use `XtRemoveWorkProc`.

```
void XtRemoveWorkProc(id)
    XtWorkProcId id;
```

id Specifies which work procedure to remove.

The `XtRemoveWorkProc` function explicitly removes the specified background work procedure.

7.9 X Event Filters

The event manager provides filters that can be applied to X user events. The filters, which screen out events that are redundant or are temporarily unwanted, handle the following:

- Pointer motion compression
- Enter/leave compression
- Exposure compression

7.9.1 Pointer Motion Compression

Widgets can have a hard time keeping up with pointer motion events. Further, they usually do not actually care about every motion event. To throw out redundant motion events, the widget class field `compress_motion` should be `True`. When a request for an event would return a motion event, the Intrinsics check if there are any other motion events immediately following the current one, and, if so, skip all but the last of them.

7.9.2 Enter/Leave Compression

To throw out pairs of enter and leave events that have no intervening events, as can happen when the user moves the pointer across a widget without stopping in it, the widget class field `compress_enterleave` should be `True`. These enter and leave events are not delivered to the client if they are found together in the input queue.

7.9.3 Exposure Compression

Many widgets prefer to process a series of exposure events as a single expose region rather than as individual rectangles. Widgets with complex displays might use the expose region as a clip list in a graphics context, and widgets with simple displays might ignore the region entirely and redisplay their whole window or might get the bounding box from the region and redisplay only that rectangle.

In either case, these widgets do not care about getting partial expose events. If the `compress_exposure` field in the widget class structure is `True`, the event manager calls the widget's expose procedure only once for each series of exposure events. In this case, all `Expose` events are accumulated into a region. When the final `Expose` event in a series (that is, the one with `count` zero) is received, the event manager replaces the rectangle in the event with the bounding box for the region and calls the widget's expose procedure, passing the modified exposure event and the region. (See the *Guide to the Xlib Library*.)

If `compress_exposure` is `False`, the event manager calls the widget's `expose` procedure for every exposure event, passing it the event and a region argument of `NULL`.

7.10 Widget Exposure and Visibility

Every primitive widget and some composite widgets display data on the screen by means of raw Xlib calls. Widgets cannot simply write to the screen and forget what they have done. They must keep enough state to redisplay the window or parts of it if a portion is obscured and then reexposed.

7.10.1 Redisplay of a Widget: the `expose` Procedure

The `expose` procedure pointer in a widget class is of type `XtExposeProc`:

```
typedef void (*XtExposeProc)(Widget, XEvent *,
                             Region);

Widget w;
XEvent *event;
Region region;
```

w Specifies the widget instance requiring redisplay.
event Specifies the exposure event giving the rectangle requiring redisplay.
region Specifies the union of all rectangles in this exposure sequence.

The redisplay of a widget upon exposure is the responsibility of the `expose` procedure in the widget's class record. If a widget has no display semantics, it can specify `NULL` for the `expose` field. Many composite widgets serve only as containers for their children and have no `expose` procedure.

Note

If the `expose` procedure is `NULL`, `XtRealizeWidget` fills in a default bit gravity of `NorthWestGravity` before it calls the widget's `realize` procedure.

If the widget's `compress_exposure` class field is `False` (see Section 7.9.3), `region` always is `NULL`. If the widget's `compress_exposure` class field is `True`, the event contains the bounding box for `region`.

A small simple widget (for example, `Label`) can ignore the bounding box information in the event and redisplay the entire window. A more complicated widget (for example, `Text`) can use the bounding box

information to minimize the amount of calculation and redisplay it does. A very complex widget uses the region as a clip list in a GC and ignores the event information. The expose procedure is responsible for exposure of all superclass data as well as its own.

However, it often is possible to anticipate the display needs of several levels of subclassing. For example, rather than separate display procedures for the widgets Label, Command, and Toggle, you could write a single display routine in Label that uses display state fields like the following:

```
Boolean invert
Boolean highlight
Dimension highlight_width
```

Label would have invert and highlight always False and highlight_width zero. Command would dynamically set highlight and highlight_width, but it would leave invert always False. Finally, Toggle would dynamically set all three. In this case, the expose procedures for Command and Toggle inherit their superclass's expose procedure. For further information, see Section 1.4.9.

7.10.2 Widget Visibility

Some widgets may use substantial computing resources to display data. However, this effort is wasted if the widget is not actually visible on the screen, that is, if the widget is obscured by another application or is iconified.

The visible field in the Core widget structure provides a hint to the widget that it need not display data. This field is guaranteed True by the time an Expose event is processed if the widget is visible but is usually False if the widget is not visible.

Widgets can use or ignore the visible hint. If they ignore it, they should have visible_interest in their widget class record set False. In such cases, the visible field is initialized True and never changes. If visible_interest is True, the event manager asks for VisibilityNotify events for the widget and updates the visible field accordingly.

7.11 X Event Handlers

Event handlers are procedures that are called when specified events occur in a widget. Most widgets need not use event handlers explicitly. Instead, they use the Intrinsics translation manager. Event handler procedure pointers are of the type XtEventHandler:

```
typedef void (*XtEventHandler)(Widget, caddr_t,
                               XEvent *);

Widget w;
caddr_t client_data;
XEvent *event;
```

w Specifies the widget for which to handle events.

client_data Specifies the client specific information registered with the event handler, which is usually NULL if the event handler is registered by the widget itself.

event Specifies the triggering event.

7.11.1 Event Handlers that Select Events

To register an event handler procedure with the dispatch mechanism, use `XtAddEventHandler`.

```
void XtAddEventHandler(w, event_mask, nonmaskable, proc,
                      client_data)

Widget w;
EventMask event_mask;
Boolean nonmaskable;
XtEventHandler proc;
caddr_t client_data;
```

w Specifies the widget for which this event handler is being registered.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies a Boolean value that indicates whether this procedure should be called on the nonmaskable events (`GraphicsExpose`, `NoExpose`, `SelectionClear`, `SelectionRequest`, `SelectionNotify`, `ClientMessage`, and `MappingNotify`).

proc Specifies the procedure that is to be called.

client_data Specifies additional data to be passed to the client's event handler.

The `XtAddEventHandler` function registers a procedure with the dispatch mechanism that is to be called when an event that matches the mask occurs on the specified widget. If the procedure is already registered with the same `client_data`, the specified mask is ORed into the existing mask. If the widget is realized, `XtAddEventHandler` calls `XSelectInput`, if necessary.

To remove a previously registered event handler, use `XtRemoveEventHandler`.

```

void XtRemoveEventHandler(w, event_mask, nonmaskable, proc,
                          client_data)

    Widget w;
    EventMask event_mask;
    Boolean nonmaskable;
    XtEventHandler proc;
    caddr_t client_data;

```

w Specifies the widget for which this procedure is registered.

event_mask Specifies the event mask for which to unregister this procedure.

nonmaskable Specifies a Boolean value that indicates whether this procedure should be removed on the nonmaskable events (GraphicsExpose, NoExpose, SelectionClear, SelectionRequest, SelectionNotify, ClientMessage, and MappingNotify).

proc Specifies the procedure that is to be removed.

client_data Specifies the client data registered.

The XtRemoveEventHandler function stops the specified procedure from receiving the specified events. The request is ignored if *client_data* does not match the value given in the call to XtAddEventHandler. If the widget is realized, XtRemoveEventHandler calls XSelectInput, if necessary. If the specified procedure has not been registered or if it has been registered with a different value of *client_data*, XtRemoveEventHandler returns without reporting an error.

To stop a procedure from receiving any events, which will remove it from the widget's *event_table* entirely, call XtRemoveEventHandler with an *event_mask* of XtAllEvents and with *nonmaskable* True.

7.11.2 Event Handlers that Do Not Select Events

On occasion, clients need to register an event handler procedure with the dispatch mechanism without causing the server to select for that event. To do this, use XtAddRawEventHandler.

```

void XtAddRawEventHandler(w, event_mask, nonmaskable, proc,
                          client_data)

    Widget w;
    EventMask event_mask;
    Boolean nonmaskable;
    XtEventHandler proc;
    caddr_t client_data;

```

w Specifies the widget for which this event handler is being registered.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies a Boolean value that indicates whether this procedure should be removed on the nonmaskable events (GraphicsExpose , NoExpose , SelectionClear , SelectionRequest , SelectionNotify , ClientMessage , and MappingNotify).

proc Specifies the procedure that is to be registered.

client_data Specifies additional data to be passed to the client's event handler.

The `XtAddRawEventHandler` function is similar to `XtAddEventHandler` except that it does not affect the widget's mask and never causes an `XSelectInput` for its events. Note that the widget might already have those mask bits set because of other nonraw event handlers registered on it.

To remove a previously registered raw event handler, use `XtRemoveRawEventHandler`.

```
void XtRemoveRawEventHandler(w, event_mask, nonmaskable,
                             proc, client_data)
```

```
Widget w;
EventMask event_mask;
Boolean nonmaskable;
XtEventHandler proc;
caddr_t client_data;
```

w Specifies the widget for which this procedure is registered.

event_mask Specifies the event mask for which to unregister this procedure.

nonmaskable Specifies a Boolean value that indicates whether this procedure should be removed on the nonmaskable events (GraphicsExpose , NoExpose , SelectionClear , SelectionRequest , SelectionNotify , ClientMessage , and MappingNotify).

proc Specifies the procedure that is to be registered.

client_data Specifies the client data registered.

The `XtRemoveRawEventHandler` function stops the specified procedure from receiving the specified events. Because the procedure is a raw event handler, this does not affect the widget's mask and never causes a call on `XSelectInput`.

7.11.3 Current Event Mask

To retrieve the event mask for a given widget, use `XtBuildEventMask`.

```
EventMask XtBuildEventMask(w)  
    Widget w;
```

w Specifies the widget.

The `XtBuildEventMask` function returns the event mask representing the logical OR of all event masks for event handlers registered on the widget with `XtAddEventHandler` and all event translations, including accelerators, installed on the widget. This is the same event mask stored into the `XSetWindowAttributes` structure by `XtRealizeWidget` and sent to the server when event handlers and translations are installed or removed on the realized widget.

Applications and other widgets (clients) often need to register a procedure with a widget that gets called under certain conditions. For example, when a widget is destroyed, every procedure on the widget's `destroy_callbacks` list is called to notify clients of the widget's impending doom.

Every widget has a `destroy_callbacks` list. Widgets can define additional callback lists as they see fit. For example, the Command widget has a callback list to notify clients when the button has been activated.

8.1 Using Callback Procedure and Callback List Definitions

Callback procedure fields for use in callback lists are of type `XtCallbackProc`:

```
typedef void (*XtCallbackProc)(Widget, caddr_t,  
                                caddr_t);  
  
    Widget w;  
    caddr_t client_data;  
    caddr_t call_data;
```

- w* Specifies the widget for which the callback is registered.
- client_data* Specifies the data that the widget should pass back to the client when the widget executes the client's callback procedure.
- call_data* Specifies any callback-specific data the widget wants to pass to the client. For example, when Scrollbar executes its `thumbChanged` callback list, it passes the new position of the thumb.

The `client_data` argument provides a way for the client registering the callback also to register client-specific data (for example, a pointer to additional information about the widget, a reason for invoking the callback, and so on). The `client_data` value should be `NULL` if all necessary information is in the widget. The `call_data` argument is a convenience to avoid having simple cases where the client could otherwise call `XtGetValues` or a widget-specific function to retrieve data from the widget. Widgets

should generally avoid putting complex state information in `call_data`. The client can use the more general data retrieval methods, if necessary.

Whenever a client wants to pass a callback list as an argument in an `XtCreateWidget`, `XtSetValues`, or `XtGetValues` call, it should specify the address of a null-terminated array of type `XtCallbackList`:

```
typedef struct {
    XtCallbackProc callback;
    caddr_t closure;
} XtCallbackRec, *XtCallbackList;
```

For example, the callback list for procedures A and B with client data `clientDataA` and `clientDataB`, respectively, is:

```
static XtCallbackRec callbacks[] = {
    {A, (caddr_t) clientDataA},
    {B, (caddr_t) clientDataB},
    {(XtCallbackProc) NULL, (caddr_t) NULL}
};
```

Although callback lists are passed by address in argument lists, the Intrinsics know about callback lists. Your widget initialize and `set_values` procedures should not allocate memory for the callback list. The Intrinsics automatically do this for you by using a different structure for their internal representation.

8.2 Identifying Callback Lists

Whenever a widget contains a callback list for use by clients, it also exports in its public `.h` file the resource name of the callback list. Applications and client widgets never access callback list fields directly. Instead, they always identify the desired callback list by using the exported resource name. All the callback manipulation functions described in this chapter check to see that the requested callback list is indeed implemented by the widget.

For the Intrinsics to find and correctly handle callback lists, they should be declared with a resource type of `XtRCallback`.

8.3 Adding Callback Procedures

To add a callback procedure to a given widget's callback list, use `XtAddCallback`.

```
void XtAddCallback(w, callback_name, callback, client_data)
    Widget w;
    String callback_name;
    XtCallbackProc callback;
    caddr_t client_data;
```

w Specifies the widget.

callback_name Specifies the callback list to which the procedure is to be appended.

callback Specifies the callback procedure.

client_data Specifies the argument that is to be passed to the specified procedure when it is invoked by XtCallCallbacks or NULL.

A callback will be invoked as many times as it occurs in the callback list.

To add a list of callback procedures to a given widget's callback list, use XtAddCallbacks.

```
void XtAddCallbacks(w, callback_name, callbacks)
    Widget w;
    String callback_name;
    XtCallbackList callbacks;
```

w Specifies the widget.

callback_name Specifies the callback list to which the procedure is to be appended.

callbacks Specifies the null-terminated list of callback procedures and corresponding client data.

8.4 Removing Callback Procedures

To delete a callback procedure from a given widget's callback list, use XtRemoveCallback.

```
void XtRemoveCallback(w, callback_name, callback, client_data)
    Widget w;
    String callback_name;
    XtCallbackProc callback;
    caddr_t client_data;
```

w Specifies the widget.

callback_name Specifies the callback list from which the procedure is to be deleted.

callback Specifies the callback procedure.
client_data Specifies the client data to match on the registered callback procedure.

The `XtRemoveCallback` function removes a callback only if both the procedure and the client data match.

To delete a list of callback procedures from a given widget's callback list, use `XtRemoveCallbacks`.

```
void XtRemoveCallbacks(w, callback_name, callbacks)  
    Widget w;  
    String callback_name;  
    XtCallbackList callbacks;
```

w Specifies the widget.
callback_name Specifies the callback list from which the procedures are to be deleted.
callbacks Specifies the null-terminated list of callback procedures and corresponding client data.

To delete all callback procedures from a given widget's callback list and free all storage associated with the callback list, use `XtRemoveAllCallbacks`.

```
void XtRemoveAllCallbacks(w, callback_name)  
    Widget w;  
    String callback_name;
```

w Specifies the widget.
callback_name Specifies the callback list to be removed.

8.5 Executing Callback Procedures

To execute the procedures in a given widget's callback list, use `XtCallCallbacks`.

```
void XtCallCallbacks(w, callback_name, call_data)  
    Widget w;  
    String callback_name;  
    caddr_t call_data;
```

w Specifies the widget.
callback_name Specifies the callback list to be executed.
call_data Specifies a callback-list specific data value to pass to each of the callback procedure in the list.

If no data is needed (for example, the `commandActivated` callback list in `Command` needs only to notify its clients that the button has been activated), the `call_data` argument can be `NULL`. The `call_data` argument is the actual data if only one (32-bit) longword is needed or is the address of the data if more than one word is needed.

8.6 Checking the Status of a Callback List

To find out the status of a given widget's callback list, use `XtHasCallbacks`.

```
typedef enum {XtCallbackNoList, XtCallbackHasNone, XtCallbackHasSome} \
XtCallbackStatus;
```

```
XtCallbackStatus XtHasCallbacks(w, callback_name)
    Widget w;
    String callback_name;
```

w Specifies the widget.

callback_name Specifies the callback list to be checked.

The `XtHasCallbacks` function first checks to see if the widget has a callback list identified by `callback_name`. If the callback list does not exist, `XtHasCallbacks` returns `XtCallbackNoList`. If the callback list exists but is empty, it returns `XtCallbackHasNone`. If the callback list exists and has at least one callback registered, it returns `XtCallbackHasSome`.

Resource Management 9

A resource is a field in the widget record with a corresponding resource entry in the resource list of the widget or any of its superclasses. This means that the field is settable by `XtCreateWidget` (by naming the field in the argument list), by an entry in the default resource files (by using either the name or class), and by `XtSetValues`. In addition, it is readable by `XtGetValues`. Not all fields in a widget record are resources. Some are for bookkeeping use by the generic routines (like `managed` and `being_destroyed`). Others can be for local bookkeeping, and still others are derived from resources (many graphics contexts and pixmaps).

Writers of widgets need to obtain a large set of resources at widget creation time. Some of the resources come from the argument list supplied in the call to `XtCreateWidget`, some from the resource database, and some from the internal defaults specified for the widget. Resources are obtained first from the argument list, then from the resource database for all resources not specified in the argument list, and lastly from the internal default, if needed.

9.1 Resource Lists

A resource entry specifies a field in the widget, the textual name and class of the field that argument lists and external resource files use to refer to the field and a default value that the field should get if no value is specified. The declaration for the `XtResource` structure is:

```
typedef struct {
    String resource_name;
    String resource_class;
    String resource_type;
    Cardinal resource_size;
    Cardinal resource_offset;
    String default_type;
    caddr_t default_address;
} XtResource, *XtResourceList;
```

The `resource_name` field contains the name used by clients to access the field in the widget. By convention, it starts with a lowercase letter and is

spelled identically to the field name, except all underscores (`_`) are deleted and the next letter is replaced by its uppercase counterpart. For example, the resource name for `background_pixel` becomes `backgroundPixel`. Widget header files typically contain a symbolic name for each resource name. All resource names, classes, and types used by the Intrinsics are named in `<Xt1/StringDefs.h>`. The Intrinsics symbolic resource names begin with `XtN` and are followed by the string name (for example, `XtNbackgroundPixel` for `backgroundPixel`).

A resource class provides two functions:

- It isolates an application from different representations that widgets can use for a similar resource.
- It lets you specify values for several actual resources with a single name. A resource class should be chosen to span a group of closely related fields.

For example, a widget can have several pixel resources: `background`, `foreground`, `border`, `block cursor`, `pointer cursor`, and so on. Typically, the background defaults to white and everything else to black. The resource class for each of these resources in the resource list should be chosen so that it takes the minimal number of entries in the resource database to make background offwhite and everything else darkblue.

In this case, the background pixel should have a resource class of `Background` and all the other pixel entries a resource class of `Foreground`. Then, the resource file needs only two lines to change all pixels to offwhite or darkblue:

```
*Background:      offwhite
*Foreground:      darkblue
```

Similarly, a widget may have several resource fonts (such as normal and bold), but all fonts should have the class `Font`. Thus, changing all fonts simply requires only a single line in the default resource file:

```
*Font: 6x13
```

By convention, resource classes are always spelled starting with a capital letter. Their symbolic names are preceded with `XtC` (for example, `XtCBackground`).

The `resource_type` field is the physical representation type of the resource. By convention, it starts with an uppercase letter and is spelled identically to the type name of the field. The resource type is used when resources are fetched to convert from the resource database format (usually `String`) or the default resource format (almost anything, but often `String`) to the desired physical representation (see Section 9.6). The Intrinsics define the following resource types:

Resource Type	Structure or Field Type
XtRAcceleratorTable	XtAccelerators
XtRBoolean	Boolean
XtRBool	Bool
XtRCallback	XtCallbackList
XtRColor	XColor
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Display*
XtRFile	FILE*
XtRFloat	float
XtRFont	Font
XtRFontStruct	XFontStruct *
XtRFunction	(*)()
XtRInt	int
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	caddr_t
XtRPosition	Position
XtRShort	short
XtRString	char*
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRWidget	Widget
XtRWindow	Window

The `resource_size` field is the size of the physical representation in bytes; you should specify it as “`sizeof(type)`” so that the compiler fills in the value. The `resource_offset` field is the offset in bytes of the field within the widget. You should use the `XtOffset` macro to retrieve this value. The `default_type` field is the representation type of the default resource value. If `default_type` is different from `resource_type` and the `default_type` is needed, the resource manager invokes a conversion procedure from `default_type` to `resource_type`. Whenever possible, the default type should be identical to the resource type in order to minimize widget creation time. However, there are sometimes no values of the type that the program can easily specify. In this case, it should be a value that the converter is guaranteed to work for (for example, `XtDefaultForeground` for a pixel resource). The `default_address` field is the address of the default resource value. The default is used if a resource is not specified in the argument list or in the resource database or if the conversion from the

representation type stored in the resource database fails, which can happen for various reasons (for example, a misspelled entry in a resource file).

Two special representation types (`XtRImmediate` and `XtRCallProc`) are usable only as default resource types. `XtRImmediate` indicates that the value in the `default_address` field is the actual value of the resource rather than the address of the value. The value must be in correct representation type for the resource. No conversion is possible since there is no source representation type. `XtRCallProc` indicates that the value in the `default_address` field is a procedure variable. This procedure is automatically invoked with the widget, `resource_offset`, and a pointer to the `XrmValue` in which to store the result and is an `XtResourceDefaultProc`:

```
typedef void (*XtResourceDefaultProc)(Widget, int,
                                      XrmValue *)

    Widget w;
    int offset;
    XrmValue *value;
```

w Specifies the widget whose resource is to be obtained.

offset Specifies the offset of the field in the widget record.

value Specifies the resource value to fill in.

The `XtResourceDefaultProc` procedure should fill in the `addr` field of the value with a pointer to the default data in its correct type.

Note

The `default_address` field in the resource structure is declared as a `caddr_t`. On some machine architectures, this may be insufficient to hold procedure variables.

To get the resource list structure for a particular class, use `XtGetResourceList`:

```
void XtGetResourceList(class, resources_return,
                      num_resources_return);

    WidgetClass class;
    XtResourceList *resources_return;
    Cardinal *num_resources_return;
```

widget_class Specifies the widget class.

resources_return

Specifies a pointer to where to store the returned resource list. The caller must free this storage using `XtFree` when done with it.

num_resources_return

Specifies a pointer to where to store the number of entries in the resource list.

If it is called before the widget class is initialized (that is, before the first widget of that class has been created), `XtGetResourceList` returns the resource list as specified in the widget class record. If it is called after the widget class has been initialized, `XtGetResourceList` returns a merged resource list that contains the resources for all superclasses.

The routines `XtSetValues` and `XtGetValues` also use the resource list to set and get widget state. For further information, see Sections 9.7.1 and 9.7.2.

Here is an abbreviated version of the resource list in the `Label` widget:

```
/* Resources specific to Label */
static XtResource resources[] = {
  {XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
   XtOffset(LabelWidget, label.foreground), XtRString, XtDefaultForeground},
  {XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct *),
   XtOffset(LabelWidget, label.font), XtRString, XtDefaultFont},
  {XtNlabel, XtCLabel, XtRString, sizeof(String),
   XtOffset(LabelWidget, label.label), XtRString, NULL},
  .
  .
  .
}
```

The complete resource name for a field of a widget instance is the concatenation of the application shell name (from `XtAppCreateShell`), the instance names of all the widget's parents up to the `ApplicationShellWidget`, the instance name of the widget itself, and the resource name of the specified field of the widget. Likewise, the full resource class of a field of a widget instance is the concatenation of the application class (from `XtAppCreateShell`), the widget class names of all the widget's parents up to the `ApplicationShellWidget` (not the superclasses), the widget class name of the widget itself, and the resource name of the specified field of the widget.

9.2 Byte Offset Calculations

To determine the byte offset of a field within a structure, use `XtOffset`.

```
Cardinal XtOffset(pointer_type, field_name)
    Type pointer_type;
    Field field_name;
```

pointer_type Specifies a type that is declared as a pointer to the structure.

field_name Specifies the name of the field for which to calculate the byte offset.

The XtOffset macro is usually used to determine the offset of various resource fields from the beginning of a widget and can be used at compile time in static initializations.

9.3 Superclass-to-Subclass Chaining of Resource Lists

The XtCreateWidget function gets resources as a superclass-to-subclass operation. That is, the resources specified in Core resource list are fetched, then those in the subclass, and so on down to the resources specified for this widget's class. Within a class, resources are fetched in the order they are declared.

In general, if a widget resource field is declared in a superclass, that field is included in the superclass's resource list and need not be included in the subclass's resource list. For example, the Core class contains a resource entry for background_pixel. Consequently, the implementation of Label need not also have a resource entry for background_pixel. However, a subclass, by specifying a resource entry for that field in its own resource list, can override the resource entry for any field declared in a superclass. This is most often done to override the defaults provided in the superclass with new ones. At class initialization time, resource lists for that class are scanned from the superclass down to the class to look for resources with the same offset. A matching resource in a subclass will be reordered to override the superclass entry. (A copy of the superclass resource list is made to avoid affecting other subclasses of the superclass.)

9.4 Subresources

A widget does not do anything to get its own resources; instead, XtCreateWidget does this automatically before calling the class initialize procedure.

Some widgets have subparts that are not widgets but for which the widget would like to fetch resources. For example, the Text widget fetches resources for its source and sink. Such widgets call XtGetSubresources to accomplish this.

```
void XtGetSubresources(w, base, name, class, resources,  
                      num_resources, args, num_args)
```

```
Widget w;  
caddr_t base;  
String name;  
String class;  
XtResourceList resources;  
Cardinal num_resources;  
ArgList args;  
Cardinal num_args;
```

w Specifies the widget that wants resources for a subpart.

base Specifies the base address of the subpart data structure where the resources should be written.

name Specifies the name of the subpart.

class Specifies the class of the subpart.

resources Specifies the resource list for the subpart.

num_resources Specifies the number of resources in the resource list.

args Specifies the argument list to override resources obtained from the resource database.

num_args Specifies the number of arguments in the argument list.

The `XtGetSubresources` function constructs a name/class list from the application name/class, the name/classes of all its ancestors, and the widget itself. Then, it appends to this list the name/class pair passed in. The resources are fetched from the argument list, the resource database, or the default values in the resource list. Then, they are copied into the subpart record. If `args` is `NULL`, `num_args` must be zero. However, if `num_args` is zero, the argument list is not referenced.

9.5 Obtaining Application Resources

To retrieve resources that are not specific to a widget but apply to the overall application, use `XtGetApplicationResources`.

```
void XtGetApplicationResources(w, base, resources,  
                               num_resources, args,  
                               num_args)
```

```
Widget w;  
caddr_t base;  
XtResourceList resources;
```

(continued on next page)

```
Cardinal num_resources;  
ArgList args;  
Cardinal num_args;
```

- w* Specifies the widget that identifies the resource database to search. (The database is that associated with the display for this widget.)
- base* Specifies the base address of the subpart data structure where the resources should be written.
- resources* Specifies the resource list for the subpart.
- num_resources* Specifies the number of resources in the resource list.
- args* Specifies the argument list to override resources obtained from the resource database.
- num_args* Specifies the number of arguments in the argument list.

The `XtGetApplicationResources` function first uses the passed widget, which is usually an application shell, to construct a resource name and class list. Then, it retrieves the resources from the argument list, the resource database, or the resource list default values. After adding base to each address, `XtGetApplicationResources` copies the resources into the address given in the resource list. If `args` is NULL, `num_args` must be zero. However, if `num_args` is zero, the argument list is not referenced. The portable way to specify application resources is to declare them as members of a structure and pass the address of the structure as the base argument.

9.6 Resource Conversions

The Intrinsic provide a mechanism for registering representation converters that are automatically invoked by the resource fetching routines. The Intrinsic additionally provide and registers several commonly used converters. This resource conversion mechanism serves several purposes:

- It permits user and application resource files to contain ASCII representations of nontextual values.
- It allows textual or other representations of default resource values that are dependent on the display, screen, or color map, and thus must be computed at run time.
- It caches all conversion source and result data. Conversions that require much computation or space (for example, string to translation table) or that require round trips to the server (for example, string to font or color) are performed only once.

9.6.1 Predefined Resource Converters

The Intrinsics define all the representations used in the Core, Composite, Constraint, and Shell widgets. It registers the following resource converters:

From XtRString to:

XtRAcceleratorTable, XtRBoolean, XtRBool, XtRCursor, XtRDimension, XtRDisplay, XtRFile, XtRFloat, XtRFont, XtRFontStruct, XtRInt, XtRPixel, XtRPosition, XtRShort, XtRTranslationTable, and XtRUnsignedChar.

From XtRColor, to: XtRPixel.

From XRInt, to:

XtRBoolean, XtRBool, XtRColor, XtRDimension, XtRFloat, XtRFont, XtRPixel, XtRPixmap, XtRPosition, XtRShort, and XtRUnsignedChar.

From XtRPixel, to: XtRColor.

The string to pixel conversion has two predefined constants that are guaranteed to work and contrast with each other (`XtDefaultForeground` and `XtDefaultBackground`). They evaluate the black and white pixel values of the widget's screen, respectively. For applications that run with reverse video, however, they evaluate the white and black pixel values of the widget's screen, respectively. Similarly, the string to font and font structure converters recognize the constant `XtDefaultFont` and evaluate this to the font in the screen's default graphics context.

9.6.2 New Resource Converters

Type converters use pointers to `XrmValue` structures (defined in `<X11/Xresource.h>`) for input and output values.

```
typedef struct {
    unsigned int size;
    caddr_t addr;
} XrmValue, *XrmValuePtr;
```

A resource converter procedure pointer is of type `XtConverter`:

```
typedef void (*XtConverter)(XrmValue *, Cardinal *,
                            XrmValue *, XrmValue *);

XrmValue *args;
Cardinal *num_args;
XrmValue *from;
XrmValue *to;
```

args Specifies a list of additional `XrmValue` arguments to the converter if additional context is needed to perform the conversion or NULL. For example, the string-to-font converter needs the widget's screen, or the string to pixel converter needs the widget's screen and color map.

num_args Specifies the number of additional `XrmValue` arguments or zero.

from Specifies the value to convert.

to Specifies the descriptor to use to return the converted value.

Type converters should perform the following actions:

- Check to see that the number of arguments passed is correct.
- Attempt the type conversion.
- If successful, return a pointer to the data in the `to` parameter; otherwise, call `XtWarningMsg` and return without modifying the `to` argument.

Most type converters just take the data described by the specified `from` argument and return data by writing into the specified `to` argument. A few need other information, which is available in the specified argument list. A type converter can invoke another type converter, which allows differing sources that may convert into a common intermediate result to make maximum use of the type converter cache.

Note that the address written `to->addr` cannot be that of a local variable of the converter because this is not valid after the converter returns. It should be a pointer to a static variable, as in the following example where `screenColor` is returned.

The following is an example of a converter that takes a string and converts it to a `Pixel`:

```
static void CvtStringToPixel(args, num_args, fromVal, toVal)
    XrmValue *args;
    Cardinal *num_args;
    XrmValue *fromVal;
    XrmValue *toVal;
{
    static XColor screenColor;
    XColor exactColor;
    Screen *screen;
    Colormap colormap;
    Status status;
    char message[1000];
    XrmQuark q;
    String params[1];
    Cardinal num_params = 1;

    if (*num_args != 2)
        XtErrorMsg("cvtStringToPixel", "wrongParameters", "XtToolkitError",
```

```

        "String to pixel conversion needs screen and colormap arguments",
        (String *)NULL, (Cardinal *)NULL);

screen = *((Screen **) args[0].addr);
colormap = *((Colormap *) args[1].addr);

LowerCase((char *) fromVal->addr, message);
q = XrmStringToQuark(message);

if (q == XtQExtdefaultbackground) { done(&screen->white_pixel, Pixel); return; }
if (q == XtQExtdefaultforeground) { done(&screen->black_pixel, Pixel); return; }

if ((char) fromVal->addr[0] == '#') { /* some color rgb definition */

    status = XParseColor(DisplayOfScreen(screen), colormap, (String) fromVal->addr,
        &screenColor);
    if (status != 0) status = XAllocColor(DisplayOfScreen(screen), colormap, &screenColor);

} else /* some color name */

    status = XAllocNamedColor(DisplayOfScreen(screen), colormap, (String) fromVal->addr,
        &screenColor, &exactColor);

if (status == 0) {

    params[0]=(String)fromVal->addr;
    XtWarningMsg("cvtStringToPixel","noColormap","XtToolkitError",
        "Cannot allocate colormap entry for \"%s\"", params, &num_params);

} else {

    toVal->addr = (caddr_t)&screenColor.pixel;
    toVal->size = sizeof(Pixel);

}
};

```

All type converters should define some set of conversion values that they are guaranteed to succeed on so these can be used in the resource defaults. This issue arises only with conversions, such as fonts and colors, where there is no string representation that all server implementations will necessarily recognize. For resources like these, the converter should define a symbolic constant (for example, `XtDefaultForeground`, `XtDefaultBackground`, or `XtDefaultFont`).

9.6.3 Issuing Conversion Warnings

The `XtStringConversionWarning` function is a convenience routine for new resource converters that convert from strings.

```
void XtStringConversionWarning(src, dst_type)
    String src, dst_type;
```

src Specifies the string that could not be converted.

dst_type Specifies the name of the type to which the string could not be converted.

The `XtStringConversionWarning` function issues a warning message with name "conversionError", type "string", class "XtToolkitError", and the default message string "Cannot convert "*src*" to type *dst_type*".

9.6.4 Registering a New Resource Converter

To register a new converter, use `XtAppAddConverter`.

```
void XtAppAddConverter(app_context, from_type, to_type,
                      converter, convert_args, num_args)
    XtAppContext app_context;
    String from_type;
    String to_type;
    XtConverter converter;
    XtConvertArgList convert_args;
    Cardinal num_args;
```

app_context Specifies the application context.

from_type Specifies the source type.

to_type Specifies the destination type.

converter Specifies the type converter procedure.

convert_args Specifies how to compute the additional arguments to the converter or NULL.

num_args Specifies the number of additional arguments to the converter or zero.

If the same *from_type* and *to_type* are specified in two calls to `XtAppAddConverter`, the second call overrides the first. For the few type converters that need additional arguments, the Intrinsics conversion mechanism provides a method of specifying how these arguments should be computed. The enumerated type `XtAddressMode` and the structure `XtConvertArgRec` specify how each argument is derived. These are defined in `<X11/Convert.h>`.

```

typedef enum {
    /* address mode                parameter representation */
    XtAddress,                    /* address */
    XtBaseOffset,                /* offset */
    XtImmediate,                 /* constant */
    XtResourceString,            /* resource name string */
    XtResourceQuark               /* resource name quark */
} XtAddressMode;

typedef struct {
    XtAddressMode address_mode;
    caddr_t address_id;
    Cardinal size;
} XtConvertArgRec, *XtConvertArgList;

```

The `address_mode` field specifies how the `address_id` field should be interpreted. `XtAddress` causes `address_id` to be interpreted as the address of the data. `XtBaseOffset` causes `address_id` to be interpreted as the offset from the widget base. `XtImmediate` causes `address_id` to be interpreted as a constant. `XtResourceString` causes `address_id` to be interpreted as the name of a resource that is to be converted into an offset from widget base. `XtResourceQuark` is an internal compiled form of an `XtResourceString`. The `size` field specifies the length of the data in bytes.

The following provides the code that was used to register the `CvtStringToPixel` routine shown earlier:

```

static XtConvertArgRec colorConvertArgs[] = {
    {XtBaseOffset, (caddr_t) XtOffset(Widget, core.screen), sizeof(Screen *)},
    {XtBaseOffset, (caddr_t) XtOffset(Widget, core.colormap), sizeof(Colormap)}
};

```

```

XtAddConverter(XtRString, XtRPixel, CvtStringToPixel,
    colorConvertArgs, XtNumber(colorConvertArgs));

```

The conversion argument descriptors `colorConvertArgs` and `screenConvertArg` are predefined. The `screenConvertArg` descriptor puts the widget's screen field into `args[0]`. The `colorConvertArgs` descriptor puts the widget's colormap field into `args[0]`, and the widget's screen field into `args[1]`.

Conversion routines should not just put a descriptor for the address of the base of the widget into `args[0]`, and use that in the routine. They should pass in the actual values that the conversion depends on. By keeping the dependencies of the conversion procedure specific, it is more likely that subsequent conversions will find what they need in the conversion cache. This way the cache is smaller and has fewer and more widely applicable entries.

9.6.5 Resource Converter Invocation

All resource-fetching routines (for example, `XtGetSubresources`, `XtGetApplicationResources`, and so on) call resource converters if the user specifies a resource that is a different representation from the desired representation or if the widget's default resource value representation is different from the desired representation.

To invoke resource conversions, use `XtConvert` or `XtDirectConvert`.

```
void XtConvert(w, from_type, from, to_type, to_return)
    Widget w;
    String from_type;
    XrmValuePtr from;
    String to_type;
    XrmValuePtr to_return;
```

w Specifies the widget to use for additional arguments (if any are needed).

from_type Specifies the source type.

from Specifies the value to be converted.

to_type Specifies the destination type.

to_return Returns the converted value.

```
void XtDirectConvert(converter, args, num_args, from, to_return)
    XtConverter converter;
    XrmValuePtr args;
    Cardinal num_args;
    XrmValuePtr from;
    XrmValuePtr to_return;
```

converter Specifies the conversion procedure that is to be called.

args Specifies the argument list that contains the additional arguments needed to perform the conversion (often NULL).

num_args Specifies the number of additional arguments (often zero).

from Specifies the value to be converted.

to_return Returns the converted value.

The `XtConvert` function looks up the type converter registered to convert `from_type` to `to_type`, computes any additional arguments needed, and then

calls `XtDirectConvert`. The `XtDirectConvert` function looks in the converter cache to see if this conversion procedure has been called with the specified arguments. If so, it returns a descriptor for information stored in the cache; otherwise, it calls the converter and enters the result in the cache. Before calling the specified converter, `XtDirectConvert` sets the return value size to zero and the return value address to `NULL`. To determine if the conversion was successful, the client should check `to_return.address` for non-`NULL`.

9.7 Reading and Writing Widget State

Any resource field in a widget can be read or written by a client. On a write operation, the widget decides what changes it will actually allow and updates all derived fields appropriately.

9.7.1 Obtaining Widget State

To retrieve the current value of a resource associated with a widget instance, use `XtGetValues`.

```
void XtGetValues(w, args, num_args)
    Widget w;
    ArgList args;
    Cardinal num_args;
```

w Specifies the widget.

args Specifies the argument list of name/address pairs that contain the resource name and the address into which the resource value is to be stored. The resource names are widget-dependent.

num_args Specifies the number of arguments in the argument list.

The `XtGetValues` function starts with the resources specified for the core widget fields and proceeds down the subclass chain to the widget. The value field of a passed argument list should contain the address into which to store the corresponding resource value. It is the caller's responsibility to allocate and deallocate this storage according to the size of the resource representation type used within the widget.

If the widget's parent is a subclass of `constraintWidgetClass`, `XtGetValues` then fetches the values for any constraint resources requested. It starts with the constraint resources specified for `constraintWidgetClass` and proceeds down to the subclass chain to the parent's constraint resources. If the argument list contains a resource name that is not found in any of the resource lists searched, the value at the corresponding address is not modified. Finally, if the `get_values_hook` procedures are non-`NULL`, they

are called in superclass-to-subclass order after all the resource values have been fetched by `XtGetValues`. This permits a subclass to provide nonwidget resource data to `XtGetValues`.

9.7.1.1 Widget Subpart Resource Data: the `get_values_hook` Procedure

– Widgets that have subparts can return resource values from them for `XtGetValues` by supplying a `get_values_hook` procedure. The `get_values_hook` procedure pointer is of type `XtArgsProc`:

```
typedef void (*XtArgsProc)(Widget, ArgList,
                          Cardinal *);

Widget w;
ArgList args;
Cardinal *num_args;
```

w Specifies the widget whose nonwidget resource values are to be retrieved.

args Specifies the argument list that was passed to `XtCreateWidget`.

num_args Specifies the number of arguments in the argument list.

The widget should call `XtGetSubvalues` and pass in its subresource list and the `arg` and `num_args` parameters.

9.7.1.2 Widget Subpart State

– To retrieve the current value of a nonwidget resource data associated with a widget instance, use `XtGetSubvalues`. For a discussion of nonwidget subclass resources, see Section 9.4.

```
void XtGetSubvalues(base, resources, num_resources, args,
                   num_args)
    caddr_t base;
    XtResourceList resources;
    Cardinal num_resources;
    ArgList args;
    Cardinal num_args;
```

base Specifies the base address of the subpart data structure where the resources should be retrieved.

resources Specifies the nonwidget resources list.

num_resources Specifies the number of resources in the resource list.

args Specifies the argument list of name/address pairs that contain the resource name and the address into which the

resource value is to be stored. The arguments and values passed in are dependent on the subpart. The storage for argument values that are pointed to by the argument list must be deallocated by the application when no longer needed.

num_args Specifies the number of arguments in the argument list.

The `XtGetSubvalues` function obtains resource values from the structure identified by `base`.

9.7.2 Setting Widget State

To modify the current value of a resource associated with a widget instance, use `XtSetValues`.

```
void XtSetValues(w, args, num_args)
    Widget w;
    ArgList args;
    Cardinal num_args;
```

w Specifies the widget.

args Specifies the argument list of name/value pairs that contain the resources to be modified and their new values. The resources and values passed are dependent on the widget being modified.

num_args Specifies the number of arguments in the argument list.

The `XtSetValues` function starts with the resources specified for the Core widget fields and proceeds down the subclass chain to the widget. At each stage, it writes the new value (if specified by one of the arguments) or the existing value (if no new value is specified) to a new widget data record. `XtSetValues` then calls the `set_values` procedures for the widget in superclass-to-subclass order. If the widget has any non-NULL `set_values_hook` fields, these are called immediately after the corresponding `set_values` procedure. This procedure permits subclasses to set nonwidget data for `XtSetValues`.

If the widget's parent is a subclass of `constraintWidgetClass`, `XtSetValues` also updates the widget's constraints. It starts with the constraint resources specified for `constraintWidgetClass` and proceeds down the subclass chain to the parent's class. At each stage, it writes the new value or the existing value to a new constraint record. It then calls the constraint `set_values` procedures from `constraintWidgetClass` down to the parent's class. The constraint `set_values` procedures are called with widget arguments, as for all `set_values` procedures, not just the constraint record arguments, so that they can make adjustments to the desired values based on full information about the widget.

XtSetValues determines if a geometry request is needed by comparing the current widget to the new widget. If any geometry changes are required, it makes the request, and the geometry manager returns XtGeometryYes, XtGeometryAlmost, or XtGeometryNo. If XtGeometryYes, XtSetValues calls the widget's resize procedure. If XtGeometryNo, XtSetValues resets the geometry fields to their original values. If XtGeometryAlmost, XtSetValues calls the set_values_almost procedure, which determines what should be done and writes new values for the geometry fields into the new widget. XtSetValues then repeats this process, deciding once more whether the geometry manager should be called.

Finally, if any of the set_values procedures returned True, XtSetValues causes the widget's expose procedure to be invoked by calling the Xlib XClearArea function on the widget's window.

9.7.2.1 Widget State: the set_values Procedure

– The set_values procedure pointer in a widget class is of type XtSetValuesFunc:

```
typedef Boolean (*XtSetValuesFunc)(Widget, Widget,
                                   Widget);

Widget current;
Widget request;
Widget new;
```

<i>current</i>	Specifies a copy of the widget as it was before the XtSetValues call.
<i>request</i>	Specifies a copy of the widget with all values changed as asked for by the XtSetValues call before any class set_values procedures have been called.
<i>new</i>	Specifies the widget with the new values that are actually allowed.

The set_values procedure should recompute any field derived from resources that are changed (for example, many GCs depend on foreground and background). If no recomputation is necessary and if none of the resources specific to a subclass require the window to be redisplayed when their values are changed, you can specify NULL for the set_values field in the class record.

Like the initialize procedure, set_values mostly deals only with the fields defined in the subclass, but it has to resolve conflicts with its superclass, especially conflicts over width and height.

Sometimes a subclass may want to overwrite values filled in by its superclass. In particular, size calculations of a superclass are often incorrect for a subclass and in this case, the subclass must modify or recalculate fields declared and computed by its superclass.

As an example, a subclass can visually surround its superclass display. In this case, the width and height calculated by the superclass `set_values` procedure are too small and need to be incremented by the size of the surround. The subclass needs to know if its superclass's size was calculated by the superclass or was specified explicitly. All widgets must place themselves into whatever size is explicitly given, but they should compute a reasonable size if no size is requested. How does a subclass know the difference between a specified size and a size computed by a superclass?

The request and new parameters provide the necessary information. The request widget is the widget as originally requested. The new widget starts with the values in the request, but it has been updated by all superclass `set_values` procedures called so far. A subclass `set_values` procedure can compare these two to resolve any potential conflicts.

In the above example, the subclass with the visual surround can see if the width and height in the request widget are zero. If so, it adds its surround size to the width and height fields in the new widget. If not, it must make do with the size originally specified.

The new widget is the actual widget instance record. Therefore, the `set_values` procedure should do all its work on the new widget (the request widget should never be modified), and if it needs to call any routines that operate on a widget, it should specify new as the widget instance.

The widget specified by new starts with the values of that specified by request but has been modified by any superclass `set_values` procedures. A widget need not refer to the request widget, unless it must resolve conflicts between the current and new widgets. Any changes that the widget needs to make, including geometry changes, should be made in the new widget.

Finally, the `set_values` procedure must return a Boolean that indicates whether the widget needs to be redisplayed. Note that a change in the geometry fields alone does not require the `set_values` procedure to return True; the X server will eventually generate an `Expose` event, if necessary. After calling all the `set_values` procedures, `XtSetValues` forces a redisplay by calling the Xlib `XClearArea` function if any of the `set_values` procedures returned True. Therefore, a `set_values` procedure should not try to do its own redisplaying.

`Set_values` procedures should not do any work in response to changes in geometry because `XtSetValues` eventually will perform a geometry request, and that request might be denied. If the widget actually changes size in

response to a `XtSetValues`, its `resize` procedure are called. Widgets should do any geometry-related work in their `resize` procedure.

Note that it is permissible to call `XtSetValues` before a widget is realized. Therefore, the `set_values` proc must not assume that the widget is realized.

9.7.2.2 Widget State: the `set_values_almost` Procedure

– The `set_values_almost` procedure pointer in a widget class is of type `XtAlmostProc`:

```
typedef void (*XtAlmostProc)(Widget, Widget,
                             XtWidgetGeometry *,
                             XtWidgetGeometry *);

Widget w;
Widget new_widget_return;
XtWidgetGeometry *request;
XtWidgetGeometry *reply;
```

w Specifies the widget on which the geometry change is requested.

new_widget_return Specifies the new widget into which the geometry changes are to be stored.

request Specifies the original geometry request that was sent to the geometry manager that returned `XtGeometryAlmost`.

reply Specifies the compromise geometry that was returned by the geometry manager that returned `XtGeometryAlmost`.

Most classes inherit this operation from their superclass by specifying `XtInheritSetValuesAlmost` in the class initialization. The Core `set_values_almost` procedure accepts the compromise suggested.

The `set_values_almost` procedure is called when a client tries to set a widget's geometry by means of a call to `XtSetValues`, and the geometry manager cannot satisfy the request but instead returns `XtGeometryAlmost` and a compromise geometry. The `set_values_almost` procedure takes the original geometry and the compromise geometry and determines whether the compromise is acceptable or a different compromise might work. It returns its results in the `new_widget` parameter, which is then sent back to the geometry manager for another try.

9.7.2.3 Widget State: the constraint `set_values` Procedure

– The constraint `set_values` procedure pointer is of type `XtSetValuesFunc`. The values passed to the parent's constraint `set_values` procedure are the same as those passed to the child's class `set_values` procedure. A class

can specify NULL for the `set_values` field of the `ConstraintPart` if it need not compute anything.

The constraint `set_values` procedure should recompute any constraint fields derived from constraint resource that are changed. Further, it should modify the widget fields as appropriate. For example, if a constraint for the maximum height of a widget is changed to a value smaller than the widget's current height, the constraint `set_values` procedure should reset the height field in the widget.

9.7.2.4 Widget Subpart State

– To set the current value of a nonwidget resource associated with a widget instance, use `XtSetSubvalues`. For a discussion of nonwidget subclass resources, see Section 9.4.

```
void XtSetSubvalues(base, resources, num_resources, args,  
                  num_args)  
    caddr_t base;  
    XtResourceList resources;  
    Cardinal num_resources;  
    ArgList args;  
    Cardinal num_args;
```

base Specifies the base address of the subpart data structure where the resources should be written.

resources Specifies the current nonwidget resources values.

num_resources Specifies the number of resources in the resource list.

args Specifies the argument list of name/value pairs that contain the resources to be modified and their new values. The resources and values passed are dependent on the subpart of the widget being modified.

num_args Specifies the number of arguments in the argument list.

The `XtSetSubvalues` function stores resources into the structure identified by `base`.

9.7.2.5 Widget Subpart Resource Data: the `set_values_hook` Procedure

– Widgets that have a subpart can set the resource values by using `XtSetValues` and supplying a `set_values_hook` procedure. The `set_values_hook` procedure pointer in a widget class is of type `XtArgsFunc`:

```
typedef Boolean (*XtArgsFunc)(Widget, Arglist,  
                             Cardinal *);  
    Widget w;  
    ArgList args;  
    Cardinal *num_args;
```

- w* Specifies the widget whose nonwidget resource values are to be changed.
- args* Specifies the argument list that was passed to XtCreateWidget.
- num_args* Specifies the number of arguments in the argument list.

Translation Management 10

Except under unusual circumstances, widgets do not hardwire the mapping of user events into widget behavior by using the event manager. Instead, they provide a default mapping of events into behavior that you can override.

The translation manager provides an interface to specify and manage the mapping of X Event sequences into widget-supplied functionality, for example, calling procedure *Abc* when the *y* key is pressed.

The translation manager uses two kinds of tables to perform translations:

- The action tables, which are in the widget class structure, specify the mapping of externally available procedure name strings to the corresponding procedure implemented by the widget class.
- A translation table, which is in the widget class structure, specifies the mapping of event sequence to procedure name strings.

You can override the translation table in the class structure for a specific widget instance by supplying a different translation table for the widget instance. The resource name is *XtNtranslations*.

10.1 Action Tables

All widget class records contain an action table. In addition, an application can register its own action tables with the translation manager so that the translation tables it provides to widget instances can access application functionality. The translation *action_proc* procedure pointer is of type *XtActionProc*:

```
typedef void (*XtActionProc)(Widget, XEvent *, String *,
                             Cardinal *);

Widget w;
XEvent *event;
String *params;
Cardinal *num_params;
```

w Specifies the widget that caused the action to be called.

event Specifies the event that caused the action to be called. If the action is called after a sequence of events, then the last event in the sequence is used.

params Specifies a pointer to the list of strings that were specified in the translation table as arguments to the action.

num_params Specifies the number of arguments specified in the translation table.

```
typedef struct _XtActionsRec {
    String action_name;
    XtActionProc action_proc;
} XtActionsRec, *XtActionList;
```

The `action_name` field is the name that you use in translation tables to access the procedure. The `action_proc` field is a pointer to a procedure that implements the functionality.

For example, the Command widget has procedures to take the following actions:

- Set the command button to indicate it is activated
- Unset the button back to its normal mode
- Highlight the button borders
- Unhighlight the button borders
- Notify any callbacks that the button has been activated

The action table for the Command widget class makes these functions available to translation tables written for Command or any subclass. The string entry is the name used in translation tables. The procedure entry (often spelled identically to the string) is the name of the C procedure that implements that function:

```
XtActionsRec actionTable[] = {
    {"Set",      Set},
    {"Unset",    Unset},
    {"Highlight", Highlight},
    {"Unhighlight", Unhighlight}
    {"Notify",   Notify},
};
```

10.1.1 Action Table Registration

To declare an action table and register it with the translation manager, use `XtAppAddActions`.

```
void XtAppAddActions(app_context, actions, num_actions)
    XtAppContext app_context;
    XtActionList actions;
    Cardinal num_actions;
```

app_context Specifies the application context.

actions Specifies the action table to register.

num_args Specifies the number of entries in this action table.

If more than one action is registered with the same name, the most recently registered action is used. If duplicate actions exist in an action table, the first is used. The Intrinsic register an action table for MenuPopup and MenuPopdown as part of XUI Toolkit initialization.

10.1.2 Action Names to Procedure Translations

The translation manager uses a simple algorithm to convert the name of a procedure specified in a translation table into the actual procedure specified in an action table. When the widget is realized, the translation manager performs a search for the name in the following tables:

- The widget's class action table for the name
- The widget's superclass action table and on up the superclass chain
- The action tables registered with XtAddActions (from the most recently added table to the oldest table)

As soon as it finds a name, the translation manager stops the search. If it cannot find a name, the translation manager generates an error.

10.2 Translation Tables

All widget instance records contain a translation table, which is a resource with no default value. A translation table specifies what action procedures are invoked for an event or a sequence of events. A translation table is a string containing a list of translations from an event sequence into one or more action procedure calls. The translations are separated from one another by newline characters (ASCII LF). The complete syntax of translation tables is specified in Appendix B.

As an example, the default behavior of Command is:

- Highlight on enter window
- Unhighlight on exit window
- Invert on left button down
- Call callbacks and reinvert on left button up

The following illustrates the Command's default translation table:

```
static String defaultTranslations =  
    " <EnterWindow>:Highlight() \n\  
    <LeaveWindow>: Unhighlight() \n\  
    <Btn1Down>:   Set() \n\  
    <Btn1Up>:     Notify() Unset()";
```

The `tm_table` field of the `CoreClass` record should be filled in at static initialization time with the string containing the class's default translations. If a class wants to inherit its superclass's translations, it can store the special value `XtInheritTranslations` into `tm_table`. After the class initialization procedures have been called, the Intrinsics compile this translation table into an efficient internal form. Then, at widget creation time, this default translation table is used for any widgets that have not had their core translations field set by the resource manager or the initialize procedures.

The resource conversion mechanism automatically compiles string translation tables that are resources. If a client uses translation tables that are not resources, it must compile them itself using `XtParseTranslationTable`.

The Intrinsics use the compiled form of the translation table to register the necessary events with the event manager. Widgets need do nothing other than specify the action and translation tables for events to be processed by the translation manager.

10.2.1 Event Sequences

An event sequence is a comma separated list of X event descriptions that describes a specific sequence of X events to map to a set of program actions. Each X event description consists of three parts:

- The X event type
- A prefix consisting of the X modifier bits
- An event specific suffix

Various abbreviations are supported to make translation tables easier to read.

10.2.2 Action Sequences

Action sequences specify what program or widget actions to take in response to incoming X events. An action sequence of action procedure call specifications. Each action procedure call consists of the name of an action procedure and a parenthesized list of string parameters to pass to that procedure.

10.3 Translation Table Management

Sometimes an application needs to destructively or nondestructively add its own translations to a widget's translation. For example, a window manager provides functions to move a window. It usually may move the window when any pointer button is pressed down in a title bar, but it allows the user to specify other translations for the middle or right button down in the title bar, and it ignores any user translations for left button down.

To accomplish this, the window manager first should create the title bar and then should merge the two translation tables into the title bar's translations. One translation table contains the translations that the window manager wants only if the user has not specified a translation for a particular event (or event sequence). The other translation table contains the translations that the window manager wants regardless of what the user has specified.

Three Intrinsics functions support this merging:

<code>XtParseTranslationTable</code>	Compiles a translation table.
<code>XtAugmentTranslations</code>	Nondestructively merges a compiled translation table into a widget's compiled translation table.
<code>XtOverrideTranslations</code>	Destructively merges a compiled translation table into a widget's compiled translation table.

To compile a translation table, use `XtParseTranslationTable`.

```
XtTranslations XtParseTranslationTable(table)  
String table;
```

table Specifies the translation table to compile.

The `XtParseTranslationTable` function compiles the translation table into the opaque internal representation of type `XtTranslations`. Note that if an empty translation table is required for any purpose, one can be obtained by calling `XtParseTranslationTable` and passing an empty string.

To merge new translations into an existing translation table, use `XtAugmentTranslations`.


```
void XtAugmentTranslations(w, translations)
    Widget w;
    XtTranslations translations;
```

w Specifies the widget into which the new translations are to be merged.

translations Specifies the compiled translation table to merge in (must not be NULL).

The `XtAugmentTranslations` function nondestructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is ignored.

To overwrite existing translations with new translations, use `XtOverrideTranslations`.

```
void XtOverrideTranslations(w, translations)
    Widget w;
    XtTranslations translations;
```

w Specifies the widget into which the new translations are to be merged.

translations Specifies the compiled translation table to merge in (must not be NULL).

The `XtOverrideTranslations` function destructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is merged in and override the widget's translation.

To replace a widget's translations completely, use `XtSetValues` on the `XtNtranslations` resource and specify a compiled translation table as the value.

To make it possible for users to easily modify translation tables in their resource files, the string-to-translation-table resource type converter allows specifying whether the table should replace, augment, or override any existing translation table in the widget. As an option, you can specify a number sign (#) as the first character of the table followed by "replace" (default), "augment", or "override" to indicate whether to replace, augment, or override any existing table.

To completely remove existing translations, use `XtUninstallTranslations`.

```
void XtUninstallTranslations(w)
    Widget w;
```

w Specifies the widget from which the translations are to be removed.

The XtUninstallTranslations function causes the entire translation table for widget to be removed.

10.4 Using Accelerators

It is often convenient to be able to bind events in one widget to actions in another. In particular, it is often useful to be able to invoke menu actions from the keyboard. The Intrinsics provide a facility, called accelerators, that let you accomplish this. An accelerator is a translation table that is bound with its actions in the context of a particular widget. The accelerator table can then be installed on some destination widget. When an action in the destination widget would cause an accelerator action to be taken, rather than causing an action in the context of the destination, the actions are executed as though triggered by an action in the accelerator widget.

Each widget instance contains that widget's exported accelerator table. Each class of widget exports a method that takes a displayable string representation of the accelerators so that widgets can display their current accelerators. The representation is the accelerator table in canonical translation table form (see Appendix B). The display_accelerator procedure pointer is of type XtStringProc:

```
typedef void (*XtStringProc)(Widget, String);
    Widget w;
    String string;
```

w Specifies the widget that the accelerators are installed on.

string Specifies the string representation of the accelerators for this widget.

Accelerators can be specified in defaults files, and the string representation is the same as for a translation table. However, the interpretation of the #augment and #override directives apply to what will happen when the accelerator is installed, that is, whether or not the accelerator translations will override the translations in the destination widget. The default is #augment, which means that the accelerator translations have lower priority than the destination translations. The #replace directive is ignored for accelerator tables.

To parse an accelerator table, use `XtParseAcceleratorTable`.

```
XtAccelerators XtParseAcceleratorTable(source)
    String source;
```

source Specifies the accelerator table to compile.

The `XtParseAcceleratorTable` function compiles the accelerator table into the opaque internal representation.

To install accelerators from a widget on another widget, use `XtInstallAccelerators`.

```
void XtInstallAccelerators(destination, source)
    Widget destination;
    Widget source;
```

destination Specifies the widget on which the accelerators are to be installed.

source Specifies the widget from which the accelerators are to come.

The `XtInstallAccelerators` function installs the accelerators from `source` onto `destination` by augmenting the destination translations with the source accelerators. If the source `display_accelerator` method is non-NULL, `XtInstallAccelerators` calls it with the source widget and a string representation of the accelerator table, which indicates that its accelerators have been installed and that it should display them appropriately. The string representation of the accelerator table is its canonical translation table representation.

As a convenience for installing all accelerators from a widget and all its descendants onto one destination, use `XtInstallAllAccelerators`.

```
void XtInstallAllAccelerators(destination, source)
    Widget destination;
    Widget source;
```

destination Specifies the widget on which the accelerators are to be installed.

source Specifies the root widget of the widget tree from which the accelerators are to come.

The `XtInstallAllAccelerators` function recursively descends the widget tree rooted at `source` and installs the accelerators of each widget encountered onto `destination`. A common use is to call `XtInstallAllAccelerators` and pass the application main window as the source.

10.5 KeyCode-to-KeySym Conversions

The translation manager provides support for automatically translating key codes in incoming key events into KeySyms. KeyCode-to-KeySym-translator procedure pointers are of type XtKeyProc:

```
typedef void (*XtKeyProc)(Display *, KeyCode,
                          Modifiers, Modifiers *,
                          KeySym *);

Display *display;
KeyCode keycode;
Modifiers modifiers;
Modifiers *modifiers_return;
KeySym *keysym_return;
```

display Specifies the display that the KeyCode is from.

keycode Specifies the KeyCode to translate.

modifiers Specifies the modifiers to the KeyCode.

modifiers_return

Returns a mask that indicates the subset of all modifiers that are examined by the key translator.

keysym_return Returns the resulting KeySym.

This procedure takes a KeyCode and modifiers and produces a KeySym. For any given key translator function, *modifiers_return* will be a constant that indicates the subset of all modifiers that are examined by the key translator.

To register a key translator, use XtSetKeyTranslator.

```
void XtSetKeyTranslator(display, proc)
    Display *display;
    XtKeyProc proc;
```

display Specifies the display from which to translate the events.

proc Specifies the procedure that is to perform key translations.

The XtSetKeyTranslator function sets the specified procedure as the current key translator. The default translator is XtTranslateKey, an XtKeyProc that uses Shift and Lock modifiers with the interpretations defined by the core protocol. It is provided so that new translators can call it to get default KeyCode-to-KeySym translations and so that the default translator can be reinstalled.

To invoke the currently registered KeyCode-to-KeySym translator, use XtTranslateKeyCode.

```

void XtTranslateKeyCode(display, keycode, modifiers,
                       modifiers_return, keysym_return)
    Display *display;
    KeyCode keycode;
    Modifiers modifiers;
    Modifiers *modifiers_return;
    KeySym *keysym_return;

```

display Specifies the display that the KeyCode is from.

keycode Specifies the KeyCode to translate.

modifiers Specifies the modifiers to the KeyCode.

modifiers_return

Returns a mask that indicates the modifiers actually used to generate the KeySym.

keysym_return Returns the resulting KeySym.

The XtTranslateKeyCode function passes the specified arguments directly to the currently registered KeyCode to KeySym translator.

To handle capitalization of nonstandard KeySyms, the Intrinsics allow clients to register case conversion routines. Case converter procedure pointers are of type XtCaseProc:

```

typedef void (*XtCaseProc)(KeySym *, KeySym *,
                           KeySym *);
    KeySym *keysym;
    KeySym *lower_return;
    KeySym *upper_return;

```

keysym Specifies the KeySym to convert.

lower_return Specifies the lowercase equivalent for the KeySym.

upper_return Specifies the uppercase equivalent for the KeySym.

If there is no case distinction, this procedure should store the KeySym into both return values.

To register a case converter, use XtRegisterCaseConverter.

```

void XtRegisterCaseConverter(display, proc, start, stop)
    Display *display;
    XtCaseProc proc;
    KeySym start;
    KeySym stop;

```

display Specifies the display from which the key events are to come.
proc Specifies the XtCaseProc that is to do the conversions.
start Specifies the first KeySym for which this converter is valid.
stop Specifies the last KeySym for which this converter is valid.

The XtRegisterCaseConverter registers the specified case converter. The start and stop arguments provide the inclusive range of KeySyms for which this converter is to be called. The new converter overrides any previous converters for KeySyms in that range. No interface exists to remove converters; you need to register an identity converter. When a new converter is registered, the Intrinsics refreshes the keyboard state if necessary. The default converter understands case conversion for all KeySyms defined in the core protocol.

To determine upper and lowercase equivalents for a KeySym, use XtConvertCase.

```
void XtConvertCase(display, keysym, lower_return, upper_return)
    Display *display;
    KeySym keysym;
    KeySym *lower_return;
    KeySym *upper_return;
```

display Specifies the display that the KeySym came from.
keysym Specifies the KeySym to convert.
lower_return Returns the lowercase equivalent of the KeySym.
upper_return Returns the uppercase equivalent of the KeySym.

The XtConvertCase function calls the appropriate converter and returns the results. A user-supplied XtKeyProc may need to use this function.

Utility Functions 11

The Intrinsics provide a number of utility functions that you can use to:

- Determine the number of elements in an array
- Translate strings to widget instances
- Manage memory usage
- Share graphics contexts
- Manipulate selections
- Merge exposure events into a region
- Translate widget coordinates
- Translate a window to a widget
- Handle errors

11.1 Determining the Number of Elements in an Array

To determine the number of elements in a fixed-size array, use `XtNumber`.

```
Cardinal XtNumber(array)  
    ArrayVariable array;
```

array Specifies a fixed-size array.

The `XtNumber` macro returns the number of elements in the specified argument lists, resources lists, and other counted arrays.

11.2 Translating Strings to Widget Instances

To translate a widget name to widget instance, use `XtNameToWidget`.

```
Widget XtNameToWidget(reference, names);  
    Widget reference;  
    String names;
```

reference Specifies the widget from which the search is to start.

names Specifies the fully qualified name of the desired widget.

The `XtNameToWidget` function looks for a widget whose name is the first component in the specified names and that is a pop-up child of reference (or a normal child if reference is a subclass of `compositeWidgetClass`). It then uses that widget as the new reference and repeats the search after deleting the first component from the specified names. If it cannot find the specified widget, `XtNameToWidget` returns `NULL`.

Note that the names argument contains the name of a widget with respect to the specified reference widget and can contain more than one widget name (separated by periods) for widgets that are not direct children of the specified reference widget.

If more than one child of the reference widget matches the name, `XtNameToWidget` can return any of the children. The Intrinsics do not require that all children of a widget have unique names. If the specified names contain more than one component and if more than one child matches the first component, `XtNameToWidget` can return `NULL` if the single branch that it follows does not contain the named widget. That is, `XtNameToWidget` does not back up and follow other matching branches of the widget tree.

11.3 Managing Memory Usage

The Intrinsics memory management functions provide uniform checking for null pointers and error reporting on memory allocation errors. These functions are completely compatible with their standard C language runtime counterparts (`malloc`, `calloc`, `realloc`, and `free`) with the following added functionality:

- `XtMalloc`, `XtCalloc`, and `XtRealloc` give an error if there is not enough memory.
- `XtFree` simply returns if passed a `NULL` pointer.
- `XtRealloc` simply allocates new storage if passed a `NULL` pointer.

See the standard C library documentation on `malloc`, `calloc`, `realloc`, and `free` for more information.

To allocate storage, use `XtMalloc`.

```
char *XtMalloc(size);
        Cardinal size;
```

size Specifies the number of bytes desired.

The `XtMalloc` functions returns a pointer to a block of storage of at least the specified size bytes. If there is insufficient memory to allocate the new block, `XtMalloc` calls `XtErrorMsg`.

To allocate and initialize an array, use `XtCalloc`.

```
char *XtCalloc(num, size);  
    Cardinal num;  
    Cardinal size;
```

num Specifies the number of array elements to allocate.

size Specifies the size of an array element in bytes.

The `XtCalloc` function allocates space for the specified number of array elements of the specified size and initializes the space to zero. If there is insufficient memory to allocate the new block, `XtCalloc` calls `XtErrorMsg`.

To change the size of an allocated block of storage, use `XtRealloc`.

```
char *XtRealloc(ptr, num);  
    char *ptr;  
    Cardinal num;
```

ptr Specifies a pointer to the old storage.

num Specifies number of bytes desired in new storage.

The `XtRealloc` function changes the size of a block of storage (possibly moving it). Then, it copies the old contents (or as much as will fit) into the new block and frees the old block. If there is insufficient memory to allocate the new block, `XtRealloc` calls `XtErrorMsg`. If `ptr` is `NULL`, `XtRealloc` allocates the new storage without copying the old contents; that is, it simply calls `XtMalloc`.

To free an allocated block of storage, use `XtFree`.

```
void XtFree(ptr);  
    char *ptr;
```

ptr Specifies a pointer to the block of storage that is to be freed.

The `XtFree` function returns storage and allows it to be reused. If `ptr` is `NULL`, `XtFree` returns immediately.

To allocate storage for a new instance of a data type, use `XtNew`.

```
type *XtNew(type);  
      type;
```

type Specifies a previously declared data type.

XtNew returns a pointer to the allocated storage. If there is insufficient memory to allocate the new block, XtNew calls XtErrorMsg. XtNew is a convenience macro that calls XtMalloc with the following arguments specified:

```
((type *) XtMalloc((unsigned) sizeof(type)))
```

To copy an instance of a string, use XtNewString.

```
String XtNewString(string);  
      String string;
```

string Specifies a previously declared string.

XtNewString returns a pointer to the allocated storage. If there is insufficient memory to allocate the new block, XtNewString calls XtErrorMsg. XtNewString is a convenience macro that calls XtMalloc with the following arguments specified:

```
(strcpy(XtMalloc((unsigned) strlen(str) + 1), str))
```

11.4 Sharing Graphics Contexts

The Intrinsics provide a mechanism whereby cooperating clients can share a graphics context (GC), thereby reducing both the number of GCs created and the total number of server calls in any given application. The mechanism is a simple caching scheme, and all GCs obtained by means of this mechanism must be treated as read-only. If a changeable GC is needed, the Xlib XCreateGC function should be used instead.

To obtain a read-only, sharable GC, use XtGetGC.

```
GC XtGetGC(w, value_mask, values)  
      Widget w;  
      XtGCMask value_mask;  
      XGCValues *values;
```

w Specifies the widget.

value_mask Specifies which fields of the values are specified.

values Specifies the actual values for this GC.

The `XtGetGC` function returns a sharable, read-only GC. The parameters to this function are the same as those for `XCreateGC` except that a widget is passed instead of a display. `XtGetGC` shares only GCs in which all values in the GC returned by `XCreateGC` are the same. In particular, it does not use the `value_mask` provided to determine which fields of the GC a widget considers relevant. The `value_mask` is used only to tell the server which fields should be filled in with widget data and which it should fill in with default values. For further information about `value_mask` and values, see `XCreateGC` in the *Guide to the Xlib Library*.

To deallocate a shared GC when it is no longer needed, use `XtReleaseGC`.

```
void XtReleaseGC(w, gc)
    Widget w;
    GC gc;
```

`w` Specifies the widget.

`gc` Specifies the GC to be deallocated.

References to sharable GCs are counted and a free request is generated to the server when the last user of a given GC destroys it.

11.5 Managing Selections

Arbitrary widgets (possibly not all in the same application) can communicate with each other by means of the XUI Toolkit global selection mechanism. The Intrinsic provide functions for providing and receiving selection data in one logical piece (atomic transfers) or in smaller logical segments (incremental transfers). Note that a physical data transfer may not necessarily correspond to the logical view.

The incremental interface is provided for a selection owner or selection client that cannot or prefers not to pass the selection value to and from the XUI Toolkit in a single block of memory. For instance, either an application that is running on a machine with limited memory may not be able to copy the entire selection value into memory at the same time, or a selection owner may already have the selection value available in discrete chunks, and it would be more efficient not to have to allocate a new block of memory to copy the pieces into it. Any owner or client that prefers to deal with the selection value in subpieces, instead of in one chunk, can use the incremental interfaces to do so. This interface between the owner/client and XUI Toolkit is independent of the underlying protocol; in the atomic interface the XUI Toolkit will break a too-large selection into smaller pieces for transport if necessary.

Note

The incremental interface is experimental and is specific to DIGITAL; applications that use it may have portability problems because it is not part of the official Intrinsics specification.

The next sections discuss how to:

- Set and get the selection timeout value
- Use atomic transfers
- Use incremental transfers

11.5.1 Setting and Getting the Selection Timeout Value

To set the Intrinsics selection timeout, use `XtAppSetSelectionTimeout`.

```
void XtAppSetSelectionTimeout(app_context, timeout)
    XtAppContext app_context;
    unsigned long timeout;
```

app_context Specifies the application context.

timeout Specifies the selection timeout in milliseconds.

To get the current selection timeout value, use `XtAppGetSelectionTimeout`.

```
unsigned long XtAppGetSelectionTimeout(app_context)
    XtAppContext app_context;
```

app_context Specifies the application context.

The `XtAppGetSelectionTimeout` function returns the current selection timeout value, in milliseconds. The selection timeout is the time within which the two communicating applications must respond to one another. The initial timeout value is set by the `selectionTimeout` application resource, or, if `selectionTimeout` is not specified, it defaults to five seconds.

11.5.2 Using Atomic Transfers

The next three sections discuss:

- Atomic transfer procedures
- Getting the selection value
- Setting the selection owner

11.5.2.1 Atomic Transfer Procedures

– The following procedures are to be used with atomic transfers. The first three are used by the selection owner, and the last one is used by the requestor.

```
typedef Boolean (*XtConvertSelectionProc)(Widget, Atom *,
                                         Atom *, Atom *,
                                         caddr_t *,
                                         unsigned long *,
                                         int *);

Widget w;
Atom *selection;
Atom *target;
Atom *type_return;
caddr_t *value_return;
unsigned long *length_return;
int *format_return;
```

- w* Specifies the widget which currently owns this selection.
- selection* Specifies the atom that describes the type of selection requested (for example, XA_PRIMARY or XA_SECONDARY).
- target* Specifies the target type of the selection that has been requested, which indicates the desired information about the selection (for example, File Name, Text, Window).
- type_return* Specifies a pointer to an atom into which the property type of the converted value of the selection is to be stored. For instance, either file name or text might have property type XA_STRING.
- value_return* Specifies a pointer into which a pointer to the converted value of the selection is to be stored. The selection owner is responsible for allocating this storage. If the selection owner has provided an XtSelectionDoneProc for the selection, this storage is owned by the selection owner; otherwise, it is owned by the Intrinsic selection mechanism, which frees it by calling XtFree when it is done with it.
- length_return* Specifies a pointer into which the number of elements in value (each of size indicated by format) is to be stored.
- format_return* Specifies a pointer into which the size in bits of the data elements of the selection value is to be stored.

This procedure is called by the Intrinsic selection mechanism to get the value of a selection as a given type from the current selection owner. It returns **True** if the owner successfully converted the selection to the target type or **False** otherwise. If the procedure returns **False** the values of the

return arguments are undefined. Each `XtConvertSelectionProc` should respond to target value `TARGETS` by returning a value containing the list of the targets they are prepared to convert their selection into.

```
typedef void (*XtLoseSelectionProc)(Widget, Atom *);
    Widget w;
    Atom *selection;
```

w Specifies the widget that has lost selection ownership.

selection Specifies the atom that describes the selection type.

This procedure is called by the Intrinsic selection mechanism to inform the specified widgets that it has lost the given selection. Note that this procedure does not ask the widget to lose the selection ownership.

```
typedef void (*XtSelectionDoneProc)(Widget, Atom *,
                                     Atom *);
    Widget w;
    Atom *selection;
    Atom *target;
```

w Specifies the widget that owns the converted selection.

selection Specifies the atom that describes the selection type that was converted.

target Specifies the target type to which the conversion was done.

This procedure is called by the Intrinsic selection mechanism to inform the selection owner when a selection requestor has successfully retrieved a selection value. If the selection owner has registered an `XtSelectionDoneProc`, it should expect it to be called once for each conversion that it performs but after the converted value has been successfully transferred to the requestor. If the selection owner has registered an `XtSelectionDoneProc`, it also owns the storage containing the converted selection value.

```
typedef void (*XtSelectionCallbackProc)(Widget, caddr_t,
                                         Atom *, Atom *,
                                         caddr_t,
                                         unsigned long *,
                                         int *);
```

(continued on next page)

```

Widget w;
caddr_t client_data;
Atom *selection;
Atom *type;
caddr_t value;
unsigned long *length;
int *format;

```

w Specifies the widget that requested the selection value.

client_data Specifies a value passed in by the widget when it requested the selection.

selection Specifies the type of selection that was requested.

type Specifies the representation type of the selection value (for example, XA_STRING). Note that it is not the target that was requested but the type that is used to represent the target. The special XUI Toolkit atom XT_CONVERT_FAIL is used to indicate that the selection conversion failed because the selection owner did not respond within the Intrinsics's selection timeout interval.

value Specifies a pointer to the selection value. The requesting client owns this storage and is responsible for freeing it by calling XtFree when it is done with it.

length Specifies the number of elements in value.

format Specifies the size in bits of the data elements of value.

This procedure is called by the Intrinsics selection mechanism to deliver the requested selection to the requestor.

11.5.2.2 Getting the Selection Value

– To obtain the selection value in a single, logical unit, use XtGetSelectionValue or XtGetSelectionValues.

```

void XtGetSelectionValue(w, selection, target, callback,
                        client_data, time)

```

```

Widget w;
Atom selection;
Atom target;
XtSelectionCallbackProc callback;
caddr_t client_data;
Time time;

```

w Specifies the widget that is making the request.

<i>selection</i>	Specifies the particular selection desired (that is, primary or secondary).
<i>target</i>	Specifies the type of the information that is needed about the selection.
<i>callback</i>	Specifies the callback procedure that is to be called when the selection value has been obtained. Note that this is how the selection value is communicated back to the client.
<i>client_data</i>	Specifies the argument that is to be passed to the specified procedure when it is called.
<i>time</i>	Specifies the timestamp that indicates when the selection is desired. This should be the timestamp of the event which triggered this request; the value <code>CurrentTime</code> is not acceptable.

The `XtGetSelectionValue` function requests the value of the selection that has been converted to the target type. The specified callback will be called some time after `XtGetSelectionValue` is called; in fact, it may be called before or after `XtGetSelectionValue` returns.

```
void XtGetSelectionValues(w, selection, targets, count,
                          callback, client_data, time)
```

```
Widget w;
Atom selection;
Atom *targets;
int count;
XtSelectionCallbackProc callback;
caddr_t client_data;
Time time;
```

<i>w</i>	Specifies the widget that is making the request.
<i>selection</i>	Specifies the particular selection desired (that is, primary or secondary).
<i>targets</i>	Specifies the types of information that is needed about the selection.
<i>count</i>	Specifies the length of the <i>targets</i> and <i>client_data</i> lists.
<i>callback</i>	Specifies the callback procedure that is to be called with each selection value obtained. Note that this is how the selection values are communicated back to the client.
<i>client_data</i>	Specifies the client data (one for each target type) that is passed to the callback procedure when it is called for that target.

time Specifies the timestamp that indicates when the selection value is desired. This should be the timestamp of the event which triggered this request; the value `CurrentTime` is not acceptable.

The `XtGetSelectionValues` function is similar to `XtGetSelectionValue` except that it takes a list of target types and a list of client data and obtains the current value of the selection converted to each of the targets. The effect is as if each target were specified in a separate call to `XtGetSelectionValue`. The callback is called once with the corresponding client data for each target. `XtGetSelectionValues` does guarantee that all the conversions will use the same selection value because the ownership of the selection cannot change in the middle of the list, as would be when calling `XtGetSelectionValue` repeatedly.

11.5.2.3 Setting the Selection Owner

– To set the selection owner when using atomic transfers, use `XtOwnSelection`.

```
Boolean XtOwnSelection(w, selection, time, convert_proc,  
                      lose_selection, done_proc)  
  
Widget w;  
Atom selection;  
Time time;  
XtConvertSelectionProc convert_proc;  
XtLoseSelectionProc lose_selection;  
XtSelectionDoneProc done_proc;
```

w Specifies the widget that wishes to become the owner.

selection Specifies an atom that describes the type of the selection (for example, `XA_PRIMARY`, `XA_SECONDARY`, or `XA_CLIPBOARD`).

time Specifies the timestamp that indicates when the selection ownership should commence. This should be the timestamp of the event that triggered ownership; the value `CurrentTime` is not acceptable.

convert_proc Specifies the procedure that is to be called whenever someone requests the current value of the selection.

lose_selection Specifies the procedure that is to be called whenever the widget has lost selection ownership or NULL if the owner is not interested in being called back.

done_proc Specifies the procedure that is called after the requestor has received the selection or NULL if the owner is not interested in being called back.

The `XtOwnSelection` function informs the Intrinsic selection mechanism that a widget believes it owns a selection. It returns `True` if the widget has successfully become the owner and `False` otherwise. The widget may fail to become the owner if some other widget has asserted ownership at a time later than this widget. Note that widgets can lose selection ownership either because someone else asserted later ownership of the selection or because the widget voluntarily gave up ownership of the selection. Also note that the `lose_selection` procedure is not called if the widget fails to obtain selection ownership in the first place.

Usually, the Intrinsic selection mechanism informs an application when one of its widgets has lost ownership of the selection. However, in response to some user actions (for example, when a user deletes the information selected), the application should explicitly inform the Intrinsic that its widget no longer is to be the selection owner by using `XtDisownSelection`.

```
void XtDisownSelection(w, selection, time)
    Widget w;
    Atom selection;
    Time time;
```

<i>w</i>	Specifies the widget that wishes to relinquish ownership.
<i>selection</i>	Specifies the atom that specifies which selection it is giving up.
<i>time</i>	Specifies the timestamp that indicates when the selection ownership is relinquished.

The `XtDisownSelection` function informs the Intrinsic selection mechanism that the specified widget is to lose ownership of the selection. If the widget does not currently own the selection either because it lost the selection or because it never had the selection to begin with, `XtDisownSelection` does nothing.

After a widget has called `XtDisownSelection`, its convert procedure is not called even if a request arrives later with a timestamp during the period that this widget owned the selection. However, its done procedure will be called if a conversion that started before the call to `XtDisownSelection` finishes after the call to `XtDisownSelection`.

11.5.3 Using Incremental Transfers

When using the incremental interface, an owner may have to processing more than one selection request for the same widget and selection value, converted to the same target, at the same time. The incremental functions take a receiver argument, which is an identifier that is guaranteed to be unique among all incremental requests that are active concurrently.

For example, consider the following:

- Upon receiving a request for the selection value, the owner sends the first segment.
- While waiting to be called to provide the next segment value but before sending it, the owner receives another request from another requestor for the same selection value.
- To distinguish between the requests, the owner uses the receiver identifier. This allows the owner to distinguish between the first requestor, who is asking for the second segment, and the second requestor, who is asking for the first segment.

The next three sections discuss:

- Incremental transfer procedures
- Getting the selection value
- Setting the selection owner

11.5.3.1 Incremental Transfer Procedures

– The following procedures are to be used with incremental transfers. The first four are used by the selection owner, and the last two are used by the requestor.

```
typedef Boolean (*XtConvertSelectionIncrProc)(Widget, Atom *,
                                             Atom *, Atom *,
                                             caddr_t *,
                                             unsigned long *,
                                             int *,
                                             unsigned long,
                                             caddr_t, caddr_t)

Widget w;
Atom *selection;
Atom *target;
Atom *type_return;
caddr_t *value_return;
int *length_return;
int *format_return;
unsigned long max_length;
caddr_t client_data;
caddr_t receiver;
```

<i>w</i>	Specifies the widget which currently owns this selection.
<i>selection</i>	Specifies the atom that describes the type of selection requested (for example, XA_PRIMARY or XA_SECONDARY).
<i>target</i>	Specifies the target type of the selection that has been requested, which indicates the sort of information about the selection which is desired (for example, File Name, Text, Window).
<i>type_return</i>	Specifies a pointer to an atom into which the property type of the converted value of the selection is to be stored. For instance, either file name or text might have property type XA_STRING.
<i>value_return</i>	Specifies a pointer into which a pointer to the converted value of the selection is to be stored. The selection owner is responsible for allocating this storage.
<i>length_return</i>	Specifies a pointer into which the number of elements in value (each of size indicated by format) is to be stored.
<i>format_return</i>	Specifies a pointer into which the size in bits of the data elements of the selection value is to be stored.
<i>max_length</i>	Specifies the network packet size.
<i>client_data</i>	Specifies the value passed in by the widget when it took ownership of the selection.
<i>receiver</i>	Specifies the ID of the receiving client.

This procedure is called by the Intrinsics whenever it needs to get the next incremental chunk of data from the selection owner. It returns True if the application has succeeded in converting the selection data or False otherwise. This procedure is called repeatedly by the Intrinsics if the owner has established selection ownership by using the XtOwnSelectionIncremental function. This procedure should store the value zero in *length_return* to indicate that all the selection has been delivered.

```
typedef void (*XtLoseSelectionIncrProc)(Widget, Atom *,
                                       caddr_t)

Widget w;
Atom *selection;
caddr_t client_data;
```

<i>w</i>	Specifies the widget that has lost the selection ownership.
<i>selection</i>	Specifies the atom that identifies the selection type.
<i>client_data</i>	Specifies the value passed in by the widget when it took ownership of the selection.

This procedure, which is optional, is called by the Intrinsics to inform the selection owner that it no longer owns the selection.

```
typedef void (*XtSelectionDoneIncrProc)(Widget, Atom *,
                                       Atom *, int,
                                       caddr_t)

Widget w;
Atom *selection;
Atom *target;
int receiver;
caddr_t client_data;
```

w Specifies the widget that owns the converted selection.

selection Specifies the atom that describes the selection type converted.

target Specifies the target type to which the conversion was done.

receiver Specifies the ID of the receiving client.

client_data Specified the value passed in by the client when it took ownership of the selection.

This procedure, which is optional, is called by the Intrinsics after each completed incremental data transfer whenever it has detected that the receiving client has retrieved the converted data. If this procedure is not used, the Intrinsics should free the allocated memory just as it would for a atomic transfer.

```
typedef void (*XtCancelConvertSelectionProc)(Widget, Atom *,
                                             Atom *, int,
                                             client_data)

Widget w;
Atom *selection;
Atom *target;
int receiver;
caddr_t client_data;
```

w Specifies the widget that owns the converted selection.

selection Specifies the atom that describes the selection type converted.

target Specifies the target type to which the conversion was done.

receiver Specifies the ID of the receiving client.

client_data Specifies the value pass in by the client when it requested the selection.

This procedure is called by the Intrinsics whenever it has determined by means of a timeout or other mechanism that the selection owner has lost the connection to the X server. Upon receiving this callback, the receiving client can free the memory and any other resources that have been allocated for the selection.

```
typedef void (*XtSelectionIncrCallbackProc)(Widget, caddr_t,
                                             Atom *, Atom*,
                                             caddr_t *,
                                             unsigned long, int)

Widget w;
caddr_t client_data;
Atom *selection;
Atom *type;
caddr_t *value;
unsigned long *length;
int *format;
```

w Specifies the widget that requested the selection value.

client_data Specifies a value passed in by the widget when it requested the selection.

selection Specifies the type of selection that was requested.

type Specifies the representation type of the selection value (for example, XA_STRING). Note that it is not the target that was requested but the type that is used to represent the target. The special XUI Toolkit atom XT_CONVERT_FAIL is used to indicate that the selection conversion failed because the selection owner did not respond within the Intrinsics's selection timeout interval.

value Specifies a pointer to the selection value. The requesting client owns this storage and is responsible for freeing it by calling XtFree when it is done with it.

length Specifies the number of elements in value.

format Specifies the size in bits of the data elements of value.

This procedure is called by the Intrinsics to deliver the next incremental chunk of data to the requestor. The data returned to the application is logically appended to all of the previously received data.

```
typedef void (*XtCancelSelectionCallbackProc)(Widget, Atom *,
                                             caddr_t)

Widget w;
Atom *selection;
caddr_t client_data;
```

- w* Specifies the widget that owns the converted selection.
- selection* Specifies the atom that describes the selection type converted.
- client_data* Specifies the value pass in by the client when it requested the selection.

This procedure is called by the Intrinsic to inform the requestor that the remainder of the selection cannot be obtained from the selection owner. Upon receiving this callback, the receiving client must determine for itself whether or not the partially completed data transfer is meaningful.

11.5.3.2 Getting the Selection Value

– To obtain the selection value when using incremental transfers, use `XtGetSelectionValueIncremental` or `XtGetSelectionValuesIncremental`.

```
void XtGetSelectionValueIncremental(w, selection, target,
                                   selection_callback,
                                   cancel_callback,
                                   client_data, time)

Widget w;
Atom selection;
Atom target;
XtSelectionIncrCallbackProc selection_callback;
XtCancelSelectionCallbackProc cancel_callback;
caddr_t client_data;
Time time;
```

- w* Specifies the widget that is making the request.
- selection* Specifies the particular selection desired (that is, primary or secondary).
- target* Specifies the type of the information that is needed about the selection.
- selection_callback* Specifies the callback procedure that is to be called to obtain the next incremental chunk of data.
- cancel_callback* Specifies the callback procedure that is to be called if the connection is lost.

client_data Specifies the argument that is to be passed to the specified procedure when it is called.

time Specifies the timestamp that indicates when the selection is desired. This should be the timestamp of the event which triggered this request; the value `CurrentTime` is not acceptable.

The `XtGetSelectionValueIncremental` function is similar to `XtGetSelectionValue` except that the callback procedure will be called repeatedly each time upon delivery of the next segment of the selection value. The end of the selection value is detected when callback is called with a value of length zero. If the transfer of the selection is aborted in the middle of a transfer, the `cancel_callback` procedure is called so that the requestor can dispose of the partial selection value it has collected up until that point.

```
void XtGetSelectionValuesIncremental(w, selection, targets,
                                     count, selection_callback,
                                     cancel_callback, client_data,
                                     time)
```

```
Widget w;
Atom selection;
Atom *targets;
int count;
XtSelectionIncrCallbackProc selection_callback;
XtCancelConvertSelectionProc cancel_callback;
caddr_t client_data;
Time time;
```

w Specifies the widget that is making the request.

selection Specifies the particular selection desired (that is, primary or secondary).

targets Specifies the types of information that is needed about the selection.

count Specifies the length of the targets and `client_data` lists.

selection_callback Specifies the callback procedure that is to be called with each selection value obtained. Note that this is how the selection values are communicated back to the client.

cancel_callback Specifies the callback procedure that is to be called when a selection request aborts because a timeout expires.

- client_data* Specifies the client data (one for each target type) that is passed to the callback procedure when it is called for that target.
- time* Specifies the timestamp that indicates when the selection value is desired. This should be the timestamp of the event which triggered this request; the value `CurrentTime` is not acceptable.

The `XtGetSelectionValuesIncremental` function is similar to `XtGetSelectionValueIncremental` except that it takes a list of targets and *client_data*. `XtGetSelectionValuesIncremental` is equivalent to calling `XtGetSelectionValueIncremental` successively for each target/*client_data* pair. `XtGetSelectionValuesIncremental` does guarantee that all the conversions will use the same selection value because the ownership of the selection cannot change in the middle of the list, as would be possible when calling `XtGetSelectionValueIncremental` repeatedly.

11.5.3.3 Setting the Selection Owner

– To set the selection owner when using incremental transfers, use `XtOwnSelectionIncremental`.

```
Boolean XtOwnSelectionIncremental(w, selection, time,
                                convert_callback,
                                lose_callback,
                                done_callback,
                                cancel_callback, client_data)

Widget w;
Atom selection;
Time time;
XtConvertSelectionIncrProc convert_callback;
XtLoseSelectionIncrProc lose_callback;
XtSelectionDoneIncrProc done_callback;
XtCancelConvertSelectionProc cancel_callback;
caddr_t client_data;
```

- w* Specifies the widget that wishes to become the owner.
- selection* Specifies an atom that describes the type of the selection (for example, `XA_PRIMARY`, `XA_SECONDARY`, or `XA_CLIPBOARD`).
- time* Specifies the timestamp that indicates when the selection ownership should commence. This should be the timestamp of the event that triggered ownership; the value `CurrentTime` is not acceptable.

- convert_proc* Specifies the procedure that is to be called whenever someone requests the current value of the selection.
- lose_selection* Specifies the procedure that is to be called whenever the widget has lost selection ownership or NULL if the owner is not interested in being called back.
- done_proc* Specifies the procedure that is called after the requestor has received the selection or NULL if the owner is not interested in being called back.
- cancel_callback* Specifies the callback procedure that is to be called when a selection request aborts because a timeout expires.
- client_data* Specifies the argument that is to be passed to the appropriate procedure when one of the condition occurs.

The `XtOwnSelectionIncremental` informs the Intrinsic incremental selection mechanism that the specified widget believes it owns the selection. It returns `True` if the specified widget successfully becomes the selection owner or `False` otherwise.

Widgets that use the incremental transfer mechanism should use `XtDisownSelection` to relinquish selection ownership.

11.6 Merging Exposure Events into a Region

The Intrinsic provide the `XtAddExposureToRegion` utility function that merges `Expose` and `GraphicsExpose` events into a region that clients can process at once rather than processing individual rectangles. (For further information about regions, see the *Guide to the Xlib Library*.)

To merge `Expose` and `GraphicsExpose` events into a region, use `XtAddExposureToRegion`.

```
void XtAddExposureToRegion(event, region)
    XEvent *event;
    Region region;
```

event Specifies a pointer to the `Expose` or `GraphicsExpose` event.

region Specifies the region object (as defined in `<X11/Xutil.h>`).

The `XtAddExposureToRegion` function computes the union of the rectangle defined by the exposure event and the specified region. Then, it stores the results back in `region`. If the event argument is not an `Expose` or `GraphicsExpose` event, `XtAddExposureToRegion` returns without an error and without modifying `region`.

This function is used by the exposure compression mechanism (see Section 7.9.3).

11.7 Translating Widget Coordinates

To translate an x-y coordinate pair from widget coordinates to root coordinates, use `XtTranslateCoords`.

```
void XtTranslateCoords(w, x, y, rootx_return, rooty_return)
    Widget w;
    Position x, y;
    Position *rootx_return, *rooty_return;
```

w Specifies the widget.

x

y Specify the widget-relative x and y coordinates.

rootx_return

rooty_return Returns the root-relative x and y coordinates.

While `XtTranslateCoords` is similar to the Xlib `XTranslateCoordinates` function, it does not generate a server request because all the required information already is in the widget's data structures.

11.8 Translating a Window to a Widget

To translate a window and display pointer into a widget instance, use `XtWindowToWidget`.

```
Widget XtWindowToWidget(display, window)
    Display *display;
    Window window;
```

display Specifies the display on which the window is defined.

window Specify the window for which you want the widget.

11.9 Handling Errors

The Intrinsics let a client register procedures that are to be called whenever a fatal or nonfatal error occurs. These facilities are intended for both error reporting and logging and for error correction or recovery.

Two levels of interface are provided:

- A high-level interface that takes an error name and class and looks the error up in an error resource database
- A low-level interface that takes a simple string

The high-level functions construct a string to pass to the lower-level interface. On UNIX-based systems, the error database usually is `/usr/lib/X11/XtErrorDB`.

Note

The application context specific error handling is not implemented on many systems. Most implementations will have just one set of error handlers. If they are set for different application contexts, the one performed last will prevail.

To obtain the error database (for example, to merge with an application or widget specific database), use `XtAppGetErrorDatabase`.

```
XrmDatabase *XtAppGetErrorDatabase(app_context)
    XtAppContext app_context;
```

app_context Specifies the application context.

The `XtAppGetErrorDatabase` function returns the address of the error database. The Intrinsics do a lazy binding of the error database and do not merge in the database file until the first call to `XtAppGetErrorDatabaseText`.

For a complete listing of all errors and warnings that can be generated by the Intrinsics, see Appendix D.

The high-level error and warning handler procedure pointers are of the type `XtErrorMsgHandler`:

```
typedef void (*XtErrorMsgHandler)(String, String,
                                   String, String,
                                   String *,
                                   Cardinal *);

String name;
String type;
String class;
String defaultp;
String *params;
Cardinal *num_params;
```

name Specifies the name that is concatenated with the specified type to form the resource name of the error message.

type Specifies the type that is concatenated with the name to form the resource name of the error message.

class Specifies the resource class of the error message.

defaultp Specifies the default message to use if no error database entry is found.

params Specifies a pointer to a list of values to be substituted in the message.

num_params Specifies the number of values in the parameter list.

The specified name can be a general kind of error, like `invalidParameters` or `invalidWindow`, and the specified type gives extra information. Standard `printf` notation is used to substitute the parameters into the message.

An error message handler can obtain the error database text for an error or a warning by calling `XtAppGetErrorDatabaseText`.

```
void XtAppGetErrorDatabaseText(app_context, name, type,  
                               class, default, buffer_return,  
                               nbytes, database)
```

```
    XtAppContext app_context;  
    char *name, *type, *class;  
    char *default;  
    char *buffer_return;  
    int nbytes;  
    XrmDatabase database;
```

app_context Specifies the application context.

name

type Specifies the name and type that are concatenated to form the resource name of the error message.

class Specifies the resource class of the error message.

default Specifies the default message to use if an error database entry is not found.

buffer_return Specifies the buffer into which the error message is to be returned.

nbytes Specifies the size of the buffer in bytes.

database Specifies the name of the alternative database that is to be used or `NULL` if the application's database is to be used.

The `XtAppGetErrorDatabaseText` returns the appropriate message from the error database or returns the specified default message if one is not found in the error database.

To register a procedure to be called on fatal error conditions, use `XtAppSetErrorMsgHandler`.

```
void XtAppSetErrorMsgHandler(app_context, msg_handler)
    XtAppContext app_context;
    XtErrorMsgHandler msg_handler;
```

app_context Specifies the application context.

msg_handler Specifies the new fatal error procedure, which should not return.

The default error handler provided by the Intrinsics constructs a string from the error resource database and calls `XtError`. Fatal error message handlers should not return. If one does, subsequent XUI Toolkit behavior is undefined.

To call the high-level error handler, use `XtAppErrorMsg`.

```
void XtAppErrorMsg(app_context, name, type, class, default,
    params, num_params)
    XtAppContext app_context;
    String name;
    String type;
    String class;
    String default;
    String *params;
    Cardinal *num_params;
```

app_context Specifies the application context.

name Specifies the general kind of error.

type Specifies the detailed name of the error.

class Specifies the resource class.

default Specifies the default message to use if an error database entry is not found.

params Specifies a pointer to a list of values to be stored in the message.

num_params Specifies the number of values in the parameter list.

The Intrinsics internal errors all have class `XtToolkitError`.

To register a procedure to be called on nonfatal error conditions, use `XtAppSetWarningMsgHandler`.

```
void XtAppSetWarningMsgHandler(app_context, msg_handler)
    XtAppContext app_context;
    XtErrorHandler msg_handler;
```

app_context Specifies the application context.

msg_handler Specifies the new nonfatal error procedure, which usually returns.

The default warning handler provided by the Intrinsics constructs a string from the error resource database and calls XtWarning.

To call the installed high-level warning handler, use XtAppWarningMsg.

```
void XtAppWarningMsg(app_context, name, type, class, default,
                    params, num_params)
    XtAppContext app_context;
    String name;
    String type;
    String class;
    String default;
    String *params;
    Cardinal *num_params;
```

app_context Specifies the application context.

name Specifies the general kind of error.

type Specifies the detailed name of the error.

class Specifies the resource class.

default Specifies the default message to use if an error database entry is not found.

params Specifies a pointer to a list of values to be stored in the message.

num_params Specifies the number of values in the parameter list.

The Intrinsics internal warnings all have class XtToolkitError.

The low-level error and warning handler procedure pointers are of type XtErrorHandler:

```
typedef void (*XtErrorHandler)(String);
    String message;
```

message Specifies the error message.

The error handler should display the message string in some appropriate fashion.

To register a procedure to be called on fatal error conditions, use `XtAppSetErrorHandler`.

```
void XtAppSetErrorHandler(app_context, handler)
    XtAppContext app_context;
    XtErrorHandler handler;
```

app_context Specifies the application context.

handler Specifies the new fatal error procedure, which should not return.

The default error handler provided by the Intrinsics is `_XtError`. On UNIX-based systems, it prints the message to standard error and terminates the application. Fatal error message handlers should not return. If one does, subsequent XUI Toolkit behavior is undefined.

To call the installed fatal error procedure, use `XtAppError`.

```
void XtAppError(app_context, message)
    XtAppContext app_context;
    String message;
```

app_context Specifies the application context.

message Specifies the message that is to be reported.

Most programs should use `XtAppErrorMsg`, not `XtAppError`, to provide for customization and internationalization of error messages.

To register a procedure to be called on nonfatal error conditions, use `XtAppSetWarningHandler`.

```
void XtAppSetWarningHandler(app_context, handler)
    XtAppContext app_context;
    XtErrorHandler handler;
```

app_context Specifies the application context.

handler Specifies the new nonfatal error procedure, which usually returns.

The default warning handler provided by the Intrinsics is `_XtWarning`. On UNIX-based systems, it prints the message to standard error and returns to the caller.

To call the installed nonfatal error procedure, use `XtAppWarning`.

```
void XtAppWarning(app_context, message)
    XtAppContext app_context;
    String message;
```

app_context Specifies the application context.

message Specifies the nonfatal error message that is to be reported.

Most programs should use `XtAppWarningMsg`, not `XtAppWarning`, to provide for customization and internationalization of warning messages.

Resource File Format A

A resource file contains text representing the default resource values for an application or set of applications. The resource file is an ASCII text file that consists of a number of lines with the following EBNF syntax:

```
resourcefile    = {line "\n"}.
line            = (comment | production).
comment        = "!" string.
production     = resourcename ":" string.
resourcename   = ["*"] name {("." | "*") name}.
string         = {<any character not including eol>}.
name           = {"A"-"Z" | "a"-"z" | "0"-"9"}.
```

If the last character on a line is a backslash (\), that line is assumed to continue on the next line.

To include a newline character in a string, use “\n”.

Translation Table Syntax B

Notation

Syntax is specified in EBNF notation with the following conventions:

- [a] Means either nothing or “a”
- { a } Means zero or more occurrences of “a”

All terminals are enclosed in double quotation masks (“ ”). Informal descriptions are enclosed in angle brackets (< >).

Syntax

The syntax of the translation table file is:

```
translationTable = [ directive ] { production }
directive       = ( "#replace" | "#override" | "#augment" ) "\n"
production     = lhs ":" rhs "\n"
lhs            = ( event | keyseq ) { "," ( event | keyseq ) }
keyseq        = "" keychar {keychar} ""
keychar       = [ "^" | "$" | "\" ] <ISO Latin 1 character>
event         = [modifier_list] "<event_type>" [ "(" count["+"] ")" ] {detail}
modifier_list = ( [!" | ":" ] {modifier} ) | "None"
modifier      = ["" ] modifier_name
count         = ("1" | "2" | "3" | "4" | ...)
modifier_name = "@" <keysym> | <see ModifierNames table below>
event_type    = <see Event Types table below>
detail        = <event specific details>
rhs           = { name "(" [params] ")" }
name          = namechar { namechar }
namechar     = { "a"-"z" | "A"-"Z" | "0"-"9" | "$" | "_" }
params       = string {"," string}.
string        = quoted_string | unquoted_string
quoted_string = "" {<Latin 1 character>} ""
unquoted_string = {<Latin 1 character except space, tab, ",", newline, ">}
```

It is often convenient to include newlines in a translation table to make it more readable. In C, indicate a newline with a “\n”:

```
" <Btn1Down>:DoSomething() \n\  
<Btn2Down>:DoSomethingElse()"
```

Modifier Names

The modifier field is used to specify normal X keyboard and button modifier mask bits. Modifiers are legal on event types `KeyPress`, `KeyRelease`, `ButtonPress`, `ButtonRelease`, `MotionNotify`, `EnterNotify`, `LeaveNotify`, and their abbreviations. An error is generated when a translation table that contains modifiers for any other events is parsed.

- If the `modifier_list` has no entries and is not “None”, it means “don’t care” on all modifiers.
- If an exclamation point (!) is specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and no other modifiers can be asserted.
- If any modifiers are specified and an exclamation point (!) is not specified, it means that the listed modifiers must be in the correct state and “don’t care” about any other modifiers.
- If a modifier is preceded by a tilde (~), it means that that modifier must not be asserted.
- If “None” is specified, it means no modifiers can be asserted.
- If a colon (:) is specified at the beginning of the modifier list, it directs the Intrinsics to apply any standard modifiers in the event to map the event keycode into a keysym. The default standard modifiers are Shift and Lock, with the interpretation as defined in the *X Window System Protocol, X Version 11*. The resulting keysym must exactly match the specified keysym, and the nonstandard modifiers in the event must match the `modifier_list`. For example, “:<Key>a” is distinct from “:<Key>A”, and “:Shift<Key>A” is distinct from “:<Key>A”.
- If a colon (:) is not specified, no standard modifiers are applied. Then, for example, “<Key>A” and “<Key>a” are equivalent.

In key sequences, a circumflex (^) is an abbreviation for the Control modifier, a dollar sign (\$) is an abbreviation for Meta, and a backslash (\) can be used to quote any character, in particular a double quote (”), a circumflex (^), a dollar sign (\$), and another backslash (\). Briefly:

No Modifiers:	None <event> detail
Any Modifiers:	<event> detail

Only these Modifiers: ! mod1 mod2 <event> detail
 These modifiers and any others: mod1 mod2 <event> detail
 The use of “None” for a modifier_list is identical to the use of and exclamation point with no modifiers.

Modifier	Abbreviation	Meaning
Ctrl	c	Control modifier bit
Shift	s	Shift modifier bit
Lock	l	Lock modifier bit
Meta	m	Meta key modifier (see below)
Hyper	h	Hyper key modifier (see below)
Super	su	Super key modifier (see below)
Alt	a	Alt key modifier (see below)
Mod1		Mod1 modifier bit
Mod2		Mod2 modifier bit
Mod3		Mod3 modifier bit
Mod4		Mod4 modifier bit
Mod5		Mod5 modifier bit
Button1		Button1 modifier bit
Button2		Button2 modifier bit
Button3		Button3 modifier bit
Button4		Button4 modifier bit
Button5		Button5 modifier bit
ANY		Any combination

A key modifier is any modifier bit whose corresponding keycode contains the corresponding left or right keysym. For example, “m” or “Meta” means any modifier bit mapping to a keycode whose keysym list contains `XK_Meta_L` or `XK_Meta_R`. Note that this interpretation is for each display, not global or even for each application context. The Control, Shift, and Lock modifier names refer explicitly to the corresponding modifier bits; there is no additional interpretation of keysyms for these modifiers.

Because it is possible to associate arbitrary keysyms with modifiers, the set of modifier key modifiers is extensible. The “@” <keysym> syntax means any modifier bit whose corresponding keycode contains the specified keysym.

A modifier_list/keysym combination in a translation matches a modifiers/keycode combination in an event in the following:

1. If a colon (:) is used, the Intrinsics call the display's `XtKeyProc` with the keycode and modifiers. To match, (`modifiers & modifiers_return`) must equal `modifier_list`, and `keysym_return` must equal the given `keysym`.
2. If (:) is not used, the Intrinsics mask off all don't-care bits from the modifiers. This value must be equal to `modifier_list`. Then, for each possible combination of don't-care modifiers in the `modifier_list`, the Intrinsics call the display's `XtKeyProc` with the keycode and that combination ORed with the cared-about modifier bits from the event. `Keysym_return` must match the `keysym` in the translation.

Event Types

The `EventType` field describes `XEvent` types. The following are the currently defined `EventType` values:

Type	Meaning
Key	KeyPress
KeyDown	
KeyUp	KeyRelease
BtnDown	ButtonPress
BtnUp	ButtonRelease
Motion	MotionNotify
PtrMoved	
MouseMoved	
Enter	EnterNotify
EnterWindow	
Leave	LeaveNotify
LeaveWindow	
FocusIn	FocusIn
FocusOut	FocusOut
Keymap	KeymapNotify
Expose	Expose
GrExp	GraphicsExpose
NoExp	NoExpose
Visible	VisibilityNotify
Create	CreateNotify
Destroy	DestroyNotify
Unmap	UnmapNotify
Map	MapNotify
MapReq	MapRequest
Reparent	ReparentNotify

Type	Meaning
Configure	ConfigureNotify
ConfigureReq	ConfigureRequest
Grav	GravityNotify
ResReq	ResizeRequest
Circ	CirculateNotify
CircReq	CirculateRequest
Prop	PropertyNotify
SelClr	SelectionClear
SelReq	SelectionRequest
Select	SelectionNotify
Clrmap	ColormapNotify
Message	ClientMessage
Mapping	MappingNotify

The supported abbreviations are:

Abbreviation	Meaning
Ctrl	KeyPress with control modifier
Meta	KeyPress with meta modifier
Shift	KeyPress with shift modifier
Btn1Down	ButtonPress with Btn1 detail
Btn1Up	ButtonRelease with Btn1 detail
Btn2Down	ButtonPress with Btn2 detail
Btn2Up	ButtonRelease with Btn2 detail
Btn3Down	ButtonPress with Btn3 detail
Btn3Up	ButtonRelease with Btn3 detail
Btn4Down	ButtonPress with Btn4 detail
Btn4Up	ButtonRelease with Btn4 detail
Btn5Down	ButtonPress with Btn5 detail
Btn5Up	ButtonRelease with Btn5 detail
BtnMotion	MotionNotify with any button modifier
Btn1Motion	MotionNotify with Button1 modifier
Btn2Motion	MotionNotify with Button2 modifier
Btn3Motion	MotionNotify with Button3 modifier
Btn4Motion	MotionNotify with Button4 modifier
Btn5Motion	MotionNotify with Button5 modifier

The Detail field is event specific and normally corresponds to the detail field of an X Event, for example, <Key>A. If no detail field is specified, then ANY is assumed.

A keysym can be specified as any of the standard keysym names, a hexadecimal number prefixed with "0x" or "0X", an octal number prefixed with "0" or a decimal number. A keysym expressed as a single digit is interpreted as the corresponding Latin 1 keysym, for example, "0" is the keysym XK_0. Other single character keysyms are treated as literal constants from Latin 1, for example, "!" is treated as 0x21. Standard keysym names are as defined in <X11/keysymdef.h> with the "XK_" prefix removed.

Canonical Representation

Every translation table has a unique, canonical text representation. This representation is passed to a widget's display_accelerator method to describe the accelerators installed on that widget. The canonical representation of a translation table file is (see also "Syntax"):

```
translationTable = { production }
production      = lhs ":" rhs "\n"
lhs             = event { "," event }
event           = [modifier_list] "<"event_type">" [ "(" count["+" ] ")" ] {detail}
modifier_list  = ["!" | ":" ] {modifier}
modifier        = [" "] modifier_name
count           = ("1" | "2" | "3" | "4" | ...)
modifier_name  = "@" <keysym> | <see canonical modifier names below>
event_type     = <see canonical event types below>
detail         = <event specific details>
rhs            = { name "(" [params] ")" }
name           = namechar { namechar }
namechar       = { "a"-"z" | "A"-"Z" | "0"-"9" | "$" | "_" }
params         = string { "," string }.
string         = quoted_string
quoted_string  = "" {<Latin 1 character>} ""
```

The canonical modifier names are:

Ctrl	Button1
Shift	Button2
Lock	Button3
Mod1	Button4
Mod2	Button5
Mod3	

Mod4
Mod5

The canonical event types are:

KeyPress	KeyRelease
ButtonPress	ButtonRelease
MotionNotify	EnterNotify
LeaveNotify	FocusIn
FocusOut	KeymapNotify
Expose	GraphicsExpose,
NoExpose	VisibilityNotify
CreateNotify	DestroyNotify
UnmapNotify	MapNotify
MapRequest	ReparentNotify
ConfigureNotify	ConfigureRequest
GravityNotify	ResizeRequest
CirculateNotify	CirculateRequest
PropertyNotify	SelectionClear
SelectionRequest	SelectionNotify
ColormapNotify	ClientMessage

Examples

- Always put more specific events in the table before more general ones:

```
Shift <Btn1Down> : twas() \n \  
<Btn1Down> : brillig()
```

- For double-click on Button 1 Up with Shift, use this specification:
Shift<Btn1Up>(2) : and()

This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down>,Shift<Btn1Up> : and()
```

- For double-click on Button 1 Down with Shift, use this specification:
Shift<Btn1Down>(2) : the()

This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down> : the()
```

- Mouse motion is always discarded when it occurs between events in a table where no motion event is specified:

```
<Btn1Down>,<Btn1Up> : slithy()
```

This is taken, even if the pointer moves a bit between the down and up events. Similarly, any motion event specified in a translation matches any number of motion events. If the motion event causes an action procedure to be invoked, the procedure is invoked after each motion event.

- If an event sequence consists of a sequence of events that is also a non-initial subsequence of another translation, it is not taken if it occurs in the context of the longer sequence. This occurs mostly in sequences like the following:

```
<Btn1Down>,<Btn1Up> : toves() \n \  
<Btn1Up> : did()
```

The second translation is taken only if the button release is not preceded by a button press or if there are intervening events between the press and the release. Be particularly aware of this when using the repeat notation, above, with buttons and keys because their expansion includes additional events, and when specifying motion events because they are implicitly included between any two other events. In particular, pointer motion and double-click translations cannot coexist in the same translation table.

- For single click on Button 1 Up with Shift and Meta, use this specification:
Shift Meta <Btn1Down>, Shift Meta<Btn1Up>: gyre()
- You can use a plus sign (+) to indicate “for any number of clicks greater than or equal to count”; for example:
Shift <Btn1Up>(2+) : and()
- To indicate EnterNotify with any modifiers, use this specification:
<Enter> : gimble()
- To indicate EnterNotify with no modifiers, use this specification:
None <Enter> : in()
- To indicate EnterNotify with Button 1 Down and Button 2 Up and don’t care about the other modifiers, use this specification:
Button1 Button2 <Enter> : the()

- To indicate EnterNotify with Button1 Down and Button2 Down exclusively, use this specification:

`! Button1 Button2 <Enter> : wabe()`

You do not need to use a tilde (~) with an exclamation point (!).

Conversion Notes C

In the X Version 10 and alpha release X Version 11 XUI Toolkit each widget class implemented an `Xt<Widget>Create` (for example, `XtLabelCreate`) function, in which most of the code was identical from widget to widget. In this XUI Toolkit, a single generic `XtCreateWidget` performs most of the common work and then calls the initialize procedure implemented for the particular widget class.

Each composite widget class also implemented the procedures `Xt<Widget>Add` and an `Xt<Widget>Delete` (for example, `XtButtonBoxAddButton` and `XtButtonBoxDeleteButton`). In the beta release X Version 11 XUI Toolkit, the composite generic procedures `XtManageChildren` and `XtUnmanageChildren` perform error-checking and screening out of certain children. Then, they call the `change_managed` procedure implemented for the widget's composite class. If the widget's parent has not yet been realized, the call on the `change_managed` procedure is delayed until realization time.

Old style calls can be implemented in the XUI Toolkit by defining one-line procedures or macros that invoke a generic routine. For example, you could define the macro `XtCreateLabel` as:

```
#define XtCreateLabel(name, parent, args, num_args) \  
    ((LabelWidget) XtCreateWidget(name, labelWidgetClass, parent, args, num_args))
```

Pop-up shells no longer automatically perform an `XtManageChild` on their child within their `insert_child` procedure. Creators of pop-up children need to call `XtManageChild` themselves.

As a convenience to people converting from earlier versions of the toolkit and for greater orthogonality, the following routines exist: `XtInitialize`, `XtMainLoop`, `XtNextEvent`, `XtProcessEvent`, `XtPeekEvent`, `XtPending`, `XtAddInput`, `XtAddTimeOut`, `XtAddWorkProc`, and `XtCreateApplicationShell`.


```
Widget XtInitialize(shell_name, application_class, options,
                   num_options, argc, argv)
    String shell_name;
    String application_class;
    XrmOptionDescRec options[];
    Cardinal num_options;
    Cardinal *argc;
    String argv[];
```

shell_name This parameter is ignored; therefore, you can specify NULL.

application_class

Specifies the class name of this application.

options Specifies how to parse the command line for any application-specific resources. The options argument is passed as a parameter to XrmParseCommand. For further information, see the *Guide to the Xlib Library*.

num_options Specifies the number of entries in options list.

argc Specifies a pointer to the number of command line parameters.

argv Specifies the command line parameters.

XtInitialize calls XtToolkitInitialize to initialize the toolkit internals, creates a default application context for use by the other convenience routines, then calls XtOpenDisplay with a display_string of NULL and an application_name of NULL, and finally calls XtAppCreateShell, with an application_name of NULL and returns the created shell. The semantics of calling XtInitialize more than once are undefined. See XtCreateApplicationContext, XtDisplayInitialize, and XtAppCreateShell for more information.

```
void XtMainLoop()
```

XtMainLoop first reads the next incoming file, timer, or X event by calling XtNextEvent. Then, it dispatches this to the appropriate registered procedure by calling XtDispatchEvent. This can be used as the main loop of XUI Toolkit applications, and, as such, it does not return. Applications are expected to exit in response to some user action. This routine has been replaced by XtAppMainLoop.

There is nothing special about XtMainLoop. It is simply an infinite loop that calls XtNextEvent then XtDispatchEvent.

```
void XtNextEvent(event_return)
    XEvent *event_return;
```

event_return Returns the event information to the specified event structure.

If no input is on the X input queue for the default application context, XtNextEvent flushes the X output buffer and waits for an event while looking at the other input sources and timeout values and calling any callback procedures triggered by them. This routine has been replaced by XtAppNextEvent. XtInitialize must be called before using this routine.

```
void XtProcessEvent(mask)
    XtInputMask mask;
```

mask Specifies the type of input to process.

XtProcessEvent processes one input event, timeout, or alternate input source (depending on the value of *mask*), waiting if necessary. It has been replaced by XtAppProcessEvent. XtInitialize must be called before using this function.

```
Boolean XtPeekEvent(event_return)
    XEvent *event_return;
```

event_return Returns the event information to the specified event structure.

If there is an event in the queue for the default application context, XtPeekEvent fills in the event and returns a non-zero value. If no X input is on the queue, XtPeekEvent flushes the output buffer and blocks until input is available, possibly calling some timeout callbacks in the process. If the input is an event, XtPeekEvent fills in the event and returns a non-zero value. Otherwise, the input is for an alternate input source, and XtPeekEvent returns zero. This routine has been replaced by XtAppPeekEvent. XtInitialize must be called before using this routine.

```
Boolean XtPending()
```

The XtPending returns a nonzero value if there are events pending from the X server or other input sources in the default application context. If there are no events pending, it flushes the output buffer and returns a zero value. It has been replaced by XtAppPending. XtInitialize must be called before using this routine.

```
XtInputId XtAddInput(source, condition, proc, client_data)
    int source;
    caddr_t condition;
    XtInputCallbackProc proc;
    caddr_t client_data;
```

source Specifies the source file descriptor on a UNIX-based system or other operating system dependent device specification.

condition Specifies the mask that indicates either a read, write, or exception condition or some operating system dependent condition.

proc Specifies the procedure that is called when input is available.

client_data Specifies the parameter to be passed to *proc* when input is available.

The `XtAddInput` function registers with the XUI Toolkit default application context a new source of events, which is usually file input but can also be file output. (The word “file” should be loosely interpreted to mean any sink or source of data.) `XtAddInput` also specifies the conditions under which the source can generate events. When input is pending on this source in the default application context, the callback procedure is called. This routine has been replaced by `XtAppAddInput`. `XtInitialize` must be called before using this routine.

```
XtIntervalId XtAddTimeOut(interval, proc, client_data)
    unsigned long interval;
    XtTimerCallbackProc proc;
    caddr_t client_data;
```

interval Specifies the time interval in milliseconds.

proc Specifies the procedure to be called when time expires.

client_data Specifies the parameter to be passed to *proc* when it is called.

The `XtAddTimeOut` function creates a timeout in the default application context and returns an identifier for it. The timeout value is set to *interval*. The callback procedure will be called after the time interval elapses, after which the timeout is removed. This routine has been replaced by `XtAppAddTimeOut`. `XtInitialize` must be called before using this routine.

```
XtWorkProcId XtAddWorkProc(proc, closure)
    XtWorkProc proc;
    Opaque closure;
```

proc Procedure to call to do the work.

closure Client data to pass to *proc* when it is called.

This routine registers a work proc in the default application context. It has been replaced by `XtAppAddWorkProc`. `XtInitialize` must be called before using this routine.

```
Widget XtCreateApplicationShell(name, widget_class, args,
                                num_args)
    String name;
    WidgetClass widget_class;
    ArgList args;
    Cardinal num_args;
```

name This parameter is ignored; therefore, you can specify NULL.

widget_class Specifies the widget class pointer for the created application shell widget. This will usually be `topLevelShellWidgetClass` or a subclass thereof.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in *args*.

`XtCreateApplicationShell` calls `XtAppCreateShell` with an `application_name` of NULL, the `application_class` passed to `XtInitialize` and the default application context created by `XtInitialize`. This routine has been replaced by `XtAppCreateShell`.

To register a new converter, use the procedure `XtAddConverter`.

```
void XtAddConverter(from_type, to_type, converter, convert_args,
                   num_args)
    String from_type;
    String to_type;
    XtConverter converter;
    XtConvertArgList convert_args;
    Cardinal num_args;
```

from_type Specifies the source type.

to_type Specifies the destination type.

converter Specifies the type converter procedure.

convert_args Specifies how to compute the additional arguments to the converter or NULL.

num_args Specifies the number of additional arguments to the converter or zero.

For the few type converters that need additional arguments, the Intrinsic conversion mechanism provides a method of specifying how these arguments should be computed. The enumerated type `XtAddressMode` and the structure `XtConvertArgRec` specify how each argument is derived. These are defined in `<X11/Convert.h>`.

```
typedef enum {
    /* address mode                parameter representation */
    XtAddress,                    /* address */
    XtBaseOffset,                /* offset */
    XtImmediate,                 /* constant */
    XtResourceString,            /* resource name string */
    XtResourceQuark              /* resource name quark */
} XtAddressMode;

typedef struct {
    XtAddressMode address_mode;
    caddr_t address_id;
    Cardinal size;
} XtConvertArgRec, *XtConvertArgList;
```

The `address_mode` field specifies how the `address_id` field should be interpreted. `XtAddress` causes `address_id` to be interpreted as the address of the data. `XtBaseOffset` causes `address_id` to be interpreted as the offset from the widget base. `XtImmediate` causes `address_id` to be interpreted as a constant. `XtResourceString` causes `address_id` to be interpreted as the name of a resource that is to be converted into an offset from widget base. `XtResourceQuark` is an internal compiled form of an `XtResourceString`. The `size` field specifies the length of the data in bytes.

The following provides the code that was used to register the `CvtStringToPixel` routine shown earlier:

```
static XtConvertArgRec colorConvertArgs[] = {
    {XtBaseOffset, (caddr_t) XtOffset(Widget, core.screen), sizeof(Screen *)},
    {XtBaseOffset, (caddr_t) XtOffset(Widget, core.colormap), sizeof(Colormap)}
};

XtAddConverter(XtRString, XtRPixel, CvtStringToPixel,
    colorConvertArgs, XtNumber(colorConvertArgs));
```

The conversion argument descriptors `colorConvertArgs` and `screenConvertArg` are predefined. The `screenConvertArg` descriptor puts the widget's `screen` field into `args[0]`. The `colorConvertArgs` descriptor puts the widget's `screen` field into `args[0]`, and the widget's `colormap` field into `args[1]`.

Conversion routines should not just put a descriptor for the address of the base of the widget into `args[0]`, and use that in the routine. They should pass in the actual values that the conversion depends on. By keeping the dependencies of the conversion procedure specific, it is more likely that subsequent conversions will find what they need in the conversion cache. This way the cache is smaller and has fewer and more widely applicable entries.

To deallocate a shared GC when it is no longer needed, use `XtDestroyGC`.

```
void XtDestroyGC(w, gc)
    Widget w;
    GC gc;
```

w Specifies the widget.

gc Specifies the GC to be deallocated.

References to sharable GCs are counted and a free request is generated to the server when the last user of a given GC destroys it. Note that some earlier versions of `XtDestroyGC` had only a `gc` argument. Therefore, this function is not very portable, and you are encouraged to use `XtReleaseGC` instead.

To declare an action table and register it with the translation manager, use `XtAddActions`.

```
void XtAddActions(actions, num_actions)
    XtActionList actions;
    Cardinal num_actions;
```

actions Specifies the action table to register.

num_args Specifies the number of entries in this action table.

If more than one action is registered with the same name, the most recently registered action is used. If duplicate actions exist in an action table, the first is used. The Intrinsics register an action table for `MenuPopup` and `MenuPopdown` as part of XUI Toolkit initialization.

To set the Intrinsics selection timeout, use `XtSetSelectionTimeout`.

```
void XtSetSelectionTimeout(timeout)
    unsigned long timeout;
```

timeout Specifies the selection timeout in milliseconds.

To get the current selection timeout value, use `XtGetSelectionTimeout`.

```
unsigned long XtGetSelectionTimeout()
```

The selection timeout is the time within which the two communicating applications must respond to one another. If one of them does not respond within this interval, the Intrinsics aborts the selection request. The default value of the selection timeout is five seconds.

To obtain the error database (for example, to merge with an application or widget specific database), use `XtGetErrorDatabase`.

```
XrmDatabase *XtGetErrorDatabase()
```

The `XtGetErrorDatabase` function returns the address of the error database. The Intrinsics do a lazy binding of the error database and do not merge in the database file until the first call to `XtGetErrorDatabaseText`.

For a complete listing of all errors and warnings that can be generated by the Intrinsics, see Appendix D.

An error message handler can obtain the error database text for an error or a warning by calling `XtGetErrorDatabaseText`.

```
void XtGetErrorDatabaseText(name, type, class, default,
                            buffer_return, nbytes)
    char *name, *type, *class;
    char *default;
    char *buffer_return;
    int nbytes;
```

name

type Specifies the name and type that are concatenated to form the resource name of the error message.

class Specifies the resource class of the error message.

default Specifies the default message to use if an error database entry is not found.

buffer_return Specifies the buffer into which the error message is to be returned.

nbytes Specifies the size of the buffer in bytes.

The `XtGetErrorDatabaseText` returns the appropriate message from the error database or returns the specified default message if one is not found in the error database.

To register a procedure to be called on fatal error conditions, use `XtSetErrorMsgHandler`.

```
void XtSetErrorMsgHandler(msg_handler)
    XtErrorMsgHandler msg_handler;
```

msg_handler Specifies the new fatal error procedure, which should not return.

The default error handler provided by the Intrinsics constructs a string from the error resource database and calls `XtError`. Fatal error message handlers should not return. If one does, subsequent XUI Toolkit behavior is undefined.

To call the high-level error handler, use `XtErrorMsg`.

```
void XtErrorMsg(name, type, class, default, params, num_params)
    String name;
    String type;
    String class;
    String default;
    String *params;
    Cardinal *num_params;
```

name Specifies the general kind of error.

type Specifies the detailed name of the error.

class Specifies the resource class.

default Specifies the default message to use if an error database entry is not found.

params Specifies a pointer to a list of values to be stored in the message.

num_params Specifies the number of values in the parameter list.

The Intrinsics internal errors all have class `XtToolkitError`.

To register a procedure to be called on nonfatal error conditions, use `XtSetWarningMsgHandler`.


```
void XtSetWarningMsgHandler(msg_handler)
    XtErrorMsgHandler msg_handler;
```

msg_handler Specifies the new nonfatal error procedure, which usually returns.

The default warning handler provided by the Intrinsics constructs a string from the error resource database and calls XtWarning.

To call the installed high-level warning handler, use XtWarningMsg.

```
void XtWarningMsg(name, type, class, default, params,
                 num_params)
    String name;
    String type;
    String class;
    String default;
    String *params;
    Cardinal *num_params;
```

name Specifies the general kind of error.

type Specifies the detailed name of the error.

class Specifies the resource class.

default Specifies the default message to use if an error database entry is not found.

params Specifies a pointer to a list of values to be stored in the message.

num_params Specifies the number of values in the parameter list.

The Intrinsics internal warnings all have class XtToolkitError.

To register a procedure to be called on fatal error conditions, use XtSetErrorHandler.

```
void XtSetErrorHandler(handler)
    XtErrorHandler handler;
```

handler Specifies the new fatal error procedure, which should not return.

The default error handler provided by the Intrinsics is `_XtError`. On UNIX-based systems, it prints the message to standard error and terminates the application. Fatal error message handlers should not return. If one does, subsequent XUI Toolkit behavior is undefined.

To call the installed fatal error procedure, use `XtError`.

```
void XtError(message)
    String message;
```

message Specifies the message that is to be reported.

Most programs should use `XtErrorMsg`, not `XtError`, to provide for customization and internationalization of error messages.

To register a procedure to be called on nonfatal error conditions, use `XtSetWarningHandler`.

```
void XtSetWarningHandler(handler)
    XtErrorHandler handler;
```

handler Specifies the new nonfatal error procedure, which usually returns.

The default warning handler provided by the Intrinsics is `_XtWarning`. On UNIX-based systems, it prints the message to standard error and returns to the caller.

To call the installed nonfatal error procedure, use `XtWarning`.

```
void XtWarning(message)
    String message;
```

message Specifies the nonfatal error message that is to be reported.

Most programs should use `XtWarningMsg`, not `XtWarning`, to provide for customization and internationalization of warning messages.

Standard Errors and Warnings D

All XUI Toolkit errors and warnings have class `XtToolkitError`. The following two tables summarize all of the errors and warnings that can be generated by the XUI Toolkit.

Error Messages

Name	Type	Default Message
<code>allocError</code>	<code>calloc</code>	Cannot perform calloc
<code>allocError</code>	<code>malloc</code>	Cannot perform malloc
<code>allocError</code>	<code>realloc</code>	Cannot perform realloc
<code>communicationError</code>	<code>select</code>	Select failed
<code>internalError</code>	<code>shell</code>	Shell's window manager interaction is broken
<code>invalidArgCount</code>	<code>xtGetValues</code>	Argument count > 0 on NULL argument list in <code>XtGetValues</code>
<code>invalidArgCount</code>	<code>xtSetValues</code>	Argument count > 0 on NULL argument list in <code>XtSetValues</code>
<code>invalidClass</code>	<code>constraintSetValue</code>	Subclass of <code>Constraint</code> required in <code>CallConstraintSetValues</code>
<code>invalidClass</code>	<code>xtAppCreateShell</code>	<code>XtAppCreateShell</code> requires non-NULL widget class
<code>invalidClass</code>	<code>xtCreatePopupShell</code>	<code>XtCreatePopupShell</code> requires non-NULL widget class
<code>invalidClass</code>	<code>xtCreateWidget</code>	<code>XtCreateWidget</code> requires non-NULL widget class
<code>invalidClass</code>	<code>xtPopdown</code>	<code>XtPopdown</code> requires a subclass of <code>shellWidgetClass</code>
<code>invalidClass</code>	<code>xtPopup</code>	<code>XtPopup</code> requires a subclass of <code>shellWidgetClass</code>
<code>invalidDisplay</code>	<code>xtInitialize</code>	Can't Open display
<code>invalidGeometryManager</code>	<code>xtMakeGeometryRequest</code>	<code>XtMakeGeometryRequest</code> - parent has no geometry manager

Name	Type	Default Message
invalidParameter	removePopupFromParent	RemovePopupFromParent requires non-NULL popuplist
invalidParameter	xtAddInput	invalid condition passed to XtAddInput
invalidParameters	xtMenuPopupAction	MenuPopup wants exactly one argument
invalidParameters	xtmenuPopdown	XtMenuPopdown called with num_params != 0 or 1
invalidParent	realize	Application shell is not a windowed widget?
invalidParent	xtCreatePopupShell	XtCreatePopupShell requires non-NULL parent
invalidParent	xtCreateWidget	XtCreateWidget requires non-NULL parent
invalidParent	xtMakeGeometryRequest	XtMakeGeometryRequest - NULL parent. Use SetValues instead
invalidParent	xtMakeGeometryRequest	XtMakeGeometryRequest - parent not composite
invalidParent	xtManageChildren	Attempt to manage a child when parent is not Composite
invalidParent	xtUnmanageChildren	Attempt to unmanage a child when parent is not Composite
invalidPopup	xtMenuPopup	Can't find popup in _XtMenuPopup
invalidPopup	xtMenuPopup	Can't find popup in _XtMenuPopup
invalidProcedure	inheritanceProc	Unresolved inheritance operation
invalidProcedure	realizeProc	No realize class procedure defined
invalidWindow	eventHandler	Event with wrong window
missingEvent	shell	Events are disappearing from under Shell
noAppContext	widgetToApplicationContext	Couldn't find ancestor with display information
noPerDisplay	closeDisplay	Couldn't find per display information
noPerDisplay	getPerDisplay	Couldn't find per display information
noSelectionProperties	freeSelectionProperty	internal error: no selection property context for display
nullProc	insertChild	NULL insert_child procedure
subclassMismatch	xtCheckSubclass	Widget class %s found when subclass of %s expected: %s

Name	Type	Default Message
translationError	mergingTablesWithCycles	Trying to merge translation tables with cycles, and can't resolve this cycle.
wrongParameters	cvtIntOrPixelToXColor	Pixel to color conversion needs screen and colormap arguments
wrongParameters	cvtStringToCursor	String to cursor conversion needs screen argument
wrongParameters	cvtStringToFont	String to font conversion needs screen argument
wrongParameters	cvtStringToFontStruct	String to cursor conversion needs screen argument
wrongParameters	cvtStringToPixel	String to pixel conversion needs screen and colormap arguments

Warning Messages

Name	Type	Default Message
ambiguousParent	xtManageChildren	Not all children have same parent in XtManageChildren
ambiguousParent	xtUnmanageChildren	Not all children have same parent in XtUnmanageChildren
communicationError	windowManager	Window Manager is confused
conversionError	string	Cannot convert string "%s" to type "%s"
displayError	invalidDisplay	Can't find display structure
grabError	grabDestroyCallback	XtAddGrab requires exclusive grab if spring_loaded is TRUE
grabError	grabDestroyCallback	XtAddGrab requires exclusive grab if spring_loaded is TRUE
grabError	xtRemoveGrab	XtRemoveGrab asked to remove a widget not on the grab list
initializationError	xtInitialize	Initializing Resource Lists twice
invalidArgCount	getResources	argument count > 0 on NULL argument list
invalidCallbackList	xtAddCallbacks	Cannot find callback list in XtAddCallbacks

Name	Type	Default Message
invalidCallbackList	xtCallCallback	Cannot find callback list in XtCallCallbacks
invalidCallbackList	xtOverrideCallback	Cannot find callback list in XtOverrideCallbacks
invalidCallbackList	xtRemoveAllCallback	Cannot find callback list in XtRemoveAllCallbacks
invalidCallbackList	xtRemoveCallbacks	Cannot find callback list in XtRemoveCallbacks
invalidChild	xtManageChildren	null child passed to XtManageChildren
invalidChild	xtUnmanageChildren	Null child passed to XtUnmanageChildren
invalidDepth	setValues	Can't change widget depth
invalidGeometry	xtMakeGeometryRequest	Shell subclass did not take care of geometry in XtSetValues
invalidParameters	compileAccelerators	String to AcceleratorTable needs no extra arguments
invalidParameters	compileTranslations	String to TranslationTable needs no extra arguments
invalidParameters	mergeTranslations	MergeTM to TranslationTable needs no extra arguments
invalidParent	xtCopyFromParent	CopyFromParent must have non-NULL parent
invalidPopup	unsupportedOperation	Pop-up menu creation is only supported on ButtonPress or EnterNotify events.
invalidPopup	unsupportedOperation	Pop-up menu creation is only supported on ButtonPress or EnterNotify events.
invalidProcedure	deleteChild	null delete_child procedure in XtDestroy
invalidProcedure	inputHandler	XtRemoveInput: Input handler not found
invalidProcedure	set_values_almost	set_values_almost procedure shouldn't be NULL
invalidResourceCount	getResources	resource count > 0 on NULL resource list
invalidResourceName	computeArgs	Cannot find resource name %s as argument to conversion
invalidShell	xtTranslateCoords	Widget has no shell ancestor

D-4 Standard Errors and Warnings

Name	Type	Default Message
invalidSizeOverride	xtDependencies	Representation size %d must match superclass's to override %s
invalidTypeOverride	xtDependencies	Representation type %s must match superclass's to override %s
invalidWidget	removePopupFromParent	RemovePopupFromParent,widget not on parent list
noColormap	cvtStringToPixel	Cannot allocate colormap entry for s
registerWindowError	xtRegisterWindow	Attempt to change already registered window.
registerWindowError	xtUnregisterWindow	Attempt to unregister invalid window.
translation error	nullTable	Can't remove accelerators from NULL table
translation error	nullTable	Tried to remove non-existent accelerators
translationError	ambiguousActions	Overriding earlier translation manager actions.
translationError	mergingNullTable	Old translation table was null, cannot modify.
translationError	nullTable	Can't translate event thorough NULL table
translationError	unboundActions	Actions not found: %s
translationError	xtTranslateInitialize	Intializing Translation manager twice.
translationParseError	parseError	translation table syntax error: %s
translationParseError	parseString	Missing ''.
typeConversionError	noConverter	No type converter registered for Widget class %s version mismatch:0idget %d vs. intrinsic %d.
versionMismatch	widget	
wrongParameters	cvtIntToBool	Integer to Bool conversion needs no extra arguments
wrongParameters	cvtIntToBoolean	Integer to Boolean conversion needs no extra arguments
wrongParameters	cvtIntToFont	Integer to Font conversion needs no extra arguments
wrongParameters	cvtIntToPixel	Integer to Pixel conversion needs no extra arguments

Name	Type	Default Message
wrongParameters	cvtIntToPixmap	Integer to Pixmap conversion needs no extra arguments
wrongParameters	cvtIntToShort	Integer to Short conversion needs no extra arguments
wrongParameters	cvtStringToBool	String to Bool conversion needs no extra arguments
wrongParameters	cvtStringToBoolean	String to Boolean conversion needs no extra arguments
wrongParameters	cvtStringToDisplay	String to Display conversion needs no extra arguments
wrongParameters	cvtStringToFile	String to File conversion needs no extra arguments
wrongParameters	cvtStringToInt	String to Integer conversion needs no extra arguments
wrongParameters	cvtStringToShort	String to Integer conversion needs no extra arguments
wrongParameters	cvtStringToUnsignedChar	String to Integer conversion needs no extra arguments
wrongParameters	cvtXColorToPixel	Color to Pixel conversion needs no extra arguments

StringDefs.h Header File E

The StringDefs.h header file contains:

```
/* Resource names */

#define XtNaccelerators      "accelerators"
#define XtNallowHoriz       "allowHoriz"
#define XtNallowVert        "allowVert"
#define XtNancestorSensitive "ancestorSensitive"
#define XtNbackground       "background"
#define XtNbackgroundPixmap "backgroundPixmap"
#define XtNborderColor      "borderColor"
#define XtNborder           "borderColor"
#define XtNborderPixmap     "borderPixmap"
#define XtNborderWidth      "borderWidth"
#define XtNcallback         "callback"
#define XtNcolormap         "colormap"
#define XtNdepth            "depth"
#define XtNdestroyCallback  "destroyCallback"
#define XtNeditType         "editType"
#define XtNfont             "font"
#define XtNforceBars        "forceBars"
#define XtNforeground       "foreground"
#define XtNfunction         "function"
#define XtNheight          "height"
#define XtNhSpace           "hSpace"
#define XtNindex            "index"
#define XtNinnerHeight      "innerHeight"
#define XtNinnerWidth       "innerWidth"
#define XtNinnerWindow      "innerWindow"
#define XtNinsertPosition   "insertPosition"
#define XtNinternalHeight   "internalHeight"
#define XtNinternalWidth    "internalWidth"
#define XtNjustify          "justify"
#define XtNknobHeight       "knobHeight"
#define XtNknobIndent       "knobIndent"
#define XtNknobPixel        "knobPixel"
```

```

#define XtNknobWidth      "knobWidth"
#define XtNlabel          "label"
#define XtNlength        "length"
#define XtNlowerRight    "lowerRight"
#define XtNmappedWhenManaged "mappedWhenManaged"
#define XtNmenuItem      "menuItem"
#define XtNname          "name"
#define XtNnotify        "notify"
#define XtNorientation   "orientation"
#define XtNparameter     "parameter"
#define XtNpopupCallback "popupCallback"
#define XtNpopdownCallback "popdownCallback"
#define XtNreverseVideo  "reverseVideo"
#define XtNscreen        "screen"
#define XtNscrollProc    "scrollProc"
#define XtNscrollDCursor "scrollDownCursor"
#define XtNscrollHCursor "scrollHorizontalCursor"
#define XtNscrollLCursor "scrollLeftCursor"
#define XtNscrollRCursor "scrollRightCursor"
#define XtNscrollUCursor "scrollUpCursor"
#define XtNscrollVCursor "scrollVerticalCursor"
#define XtNselection     "selection"
#define XtNselectionArray "selectionArray"
#define XtNsensitive     "sensitive"
#define XtNshown        "shown"
#define XtNspace        "space"
#define XtNstring       "string"
#define XtNtextOptions  "textOptions"
#define XtNtextSink     "textSink"
#define XtNtextSource   "textSource"
#define XtNthickness    "thickness"
#define XtNthumb        "thumb"
#define XtNthumbProc    "thumbProc"
#define XtNtop          "top"
#define XtNtranslations "translations"
#define XtNuseBottom    "useBottom"
#define XtNuseRight     "useRight"
#define XtNvalue        "value"
#define XtNvSpace       "vSpace"
#define XtNwidth        "width"
#define XtNwindow       "window"
#define XtNx            "x"
#define XtNy            "y"

```

```

/* Class types */

#define XtCAccelerators      "Accelerators"
#define XtCBackground      "Background"
#define XtCBoolean         "Boolean"
#define XtCBorderColor     "BorderColor"
#define XtCBorderWidth    "BorderWidth"
#define XtCCallback        "Callback"
#define XtCColormap        "Colormap"
#define XtCColor           "Color"
#define XtCCursor          "Cursor"
#define XtCDepth           "Depth"
#define XtCEditType        "EditType"
#define XtCEventBindings   "EventBindings"
#define XtCFile            "File"
#define XtCFont            "Font"
#define XtCForeground      "Foreground"
#define XtCFraction        "Fraction"
#define XtCFunction        "Function"
#define XtCHeight          "Height"
#define XtCHSpace          "HSpace"
#define XtCIndex           "Index"
#define XtCInterval        "Interval"
#define XtCJustify         "Justify"
#define XtCKnobIndent      "KnobIndent"
#define XtCKnobPixel       "KnobPixel"
#define XtCLabel           "Label"
#define XtCLength          "Length"
#define XtCMappedWhenManaged "MappedWhenManaged"
#define XtCMargin          "Margin"
#define XtCMenuEntry       "MenuEntry"
#define XtCNotify          "Notify"
#define XtCOrientation     "Orientation"
#define XtCParameter       "Parameter"
#define XtCPixmap          "Pixmap"
#define XtCPosition        "Position"
#define XtCScreen          "Screen"
#define XtCScrollProc      "ScrollProc"
#define XtCScrollDCursor   "ScrollDownCursor"
#define XtCScrollHCursor   "ScrollHorizontalCursor"
#define XtCScrollLCursor   "ScrollLeftCursor"
#define XtCScrollRCursor   "ScrollRightCursor"
#define XtCScrollUCursor   "ScrollUpCursor"
#define XtCScrollVCursor   "ScrollVerticalCursor"

```

```

#define XtCSelection          "Selection"
#define XtCSensitive         "Sensitive"
#define XtCSelectionArray    "SelectionArray"
#define XtCSpace             "Space"
#define XtCString            "String"
#define XtCTextOptions       "TextOptions"
#define XtCTextPosition      "TextPosition"
#define XtCTextSink          "TextSink"
#define XtCTextSource        "TextSource"
#define XtCThickness         "Thickness"
#define XtCThumb             "Thumb"
#define XtCTranslations      "Translations"
#define XtCValue             "Value"
#define XtCVSpace            "VSpace"
#define XtCWidth             "Width"
#define XtCWindow            "Window"
#define XtCX                  "X"
#define XtCY                  "Y"

/* Representation types */

#define XtRAcceleratorTable  "AcceleratorTable"
#define XtRBoolean           "Boolean"
#define XtRCallback          "Callback"
#define XtRCallProc          "CallProc"
#define XtRColor             "Color"
#define XtRCursor            "Cursor"
#define XtRDimension         "Dimension"
#define XtRDisplay           "Display"
#define XtREditMode          "EditMode"
#define XtRFile              "File"
#define XtRFont              "Font"
#define XtRFontStruct        "FontStruct"
#define XtRFunction           "Function"
#define XtRGeometry          "Geometry"
#define XtRImmediate         "Immediate"
#define XtRInt               "Int"
#define XtRJustify           "Justify"
#define XtRLongBoolean       "LongBoolean"
#define XtROrientation        "Orientation"
#define XtRPixel             "Pixel"
#define XtRPixmap            "Pixmap"
#define XtRPointer           "Pointer"
#define XtRPosition          "Position"

```

```

#define XtRShort           "Short"
#define XtRString         "String"
#define XtRStringTable    "StringTable"
#define XtRUnsignedChar   "UnsignedChar"
#define XtRTranslationTable "TranslationTable"
#define XtRWindow         "Window"

/* Boolean enumeration constants */

#define XtEoff            "off"
#define XtEfalse         "false"
#define XtEno            "no"
#define XtEon            "on"
#define XtEtrue          "true"
#define XtEyes           "yes"

/* Orientation enumeration constants */

#define XtEvertical      "vertical"
#define XtEhorizontal    "horizontal"

/* text edit enumeration constants */

#define XtEtextRead      "read"
#define XtEtextAppend    "append"
#define XtEtextEdit      "edit"

/* color enumeration constants */

#define XtExtdefaultbackground "xtdefaultbackground"
#define XtExtdefaultforeground "xtdefaultforeground"

/* font constant */

#define XtExtdefaultfont   "xtdefaultfont"

```


Index

/

`/usr/lib/X11/app-defaults/`, 2-6
`/usr/lib/X11/XtErrorDB`, 11-21

A

Above, 6-4
Accelerator, 10-7
accept_focus procedure, 7-7
Action Table, 10-2
action_proc procedure, 10-1
application context, 2-2
Application programmer, 1-2
Application, 4-9
ApplicationShell, 4-1, 4-2
ApplicationShellWidget, 4-4, 9-5
applicationShellWidgetClass, 4-4
ApplicationShellWidgetClass, 4-5
ArgList, 1-13, 2-10, 2-11

B

Background, 9-2
Below, 6-4
BottomIf, 6-4
ButtonPress, 5-6, 7-4, 7-10, B-2,
B-4, B-5, B-7
ButtonRelease, 7-4, 7-10, B-2, B-4,

B-5, B-7

C

calloc, 11-2
CenterGravity, 2-19
Chaining, 2-14, 2-16, 9-6
 Subclass, 1-21
 superclass, 1-21
change_managed procedure, 3-4
CirculateNotify, B-4, B-7
CirculateRequest, B-4, B-7
Class Initialization, 1-22
Class, 1-2
class_initialize procedure, 1-22
class_name, 1-17
Client, 1-2
ClientMessage, 7-16, 7-17, 7-18, B-4,
B-7
ColormapNotify, B-4, B-7
Composite widgets, 3-1
Composite, 1-9, 1-10, 1-11, 1-23, 1-24,
1-25, 2-1, 3-2, 4-2, 5-1, 6-1, 6-2,
7-6, 9-9
CompositeClassPart, 1-9
CompositeClassRec, 1-10
CompositePart, 1-9, 1-10, 1-12
CompositeWidget, 1-10
CompositeWidgetClass, 1-10
compositeWidgetClass, 1-10, 2-12,
2-17
CompositeWidgetClass, 2-19
compositeWidgetClass, 2-19, 2-23,

2-24, 3-1, 3-2, 3-4, 3-6, 3-8, 4-2, 6-3, 11-2
 compress_enterleave, 7-13
 compress_expose field, 7-13
 compress_motion, 7-13
 Configure Window, 6-1
 ConfigureNotify, 2-21, 3-2, B-4, B-7
 ConfigureRequest, B-4, B-7
 ConstrainP.h, 1-17
 Constraint.h, 1-15
 Constraint, 1-11, 1-12, 1-15, 1-17, 1-22, 3-2, 3-8, 3-9, 9-9
 ConstraintClassPart, 1-11, 1-22, 2-16, 2-25, 3-9
 ConstraintClassRec, 1-11
 ConstraintPart, 1-11, 1-12, 9-21
 ConstraintWidget, 1-11, 1-12
 ConstraintWidgetClass, 1-11
 constraintWidgetClass, 1-11
 ConstraintWidgetClass, 1-12
 constraintWidgetClass, 2-12
 ConstraintWidgetClass, 2-19
 constraintWidgetClass, 2-23, 2-25, 3-8, 9-16, 9-18
 CopyFromParent, 2-19, 2-20
 Core, 1-6, 1-8, 1-9, 1-12, 1-22, 1-23, 1-24, 1-25, 2-12, 2-17, 2-18, 2-19, 2-20, 3-5, 7-15, 9-17, 9-6, 9-9, 9-20
 CoreClass, 10-4
 CoreClassPart, 1-6, 2-24
 CorePart, 1-6, 1-7, 1-10, 5-1
 CreateNotify, B-4, B-7
 CurrentTime, 11-10, 11-11, 11-18, 11-19
 CWBorderWidth, 6-4
 CWHeight, 6-4
 CWSibling, 6-4
 CWStackMode, 6-4, 6-11
 CWWidth, 6-4
 CWX, 6-4

CWY, 6-4

D

delete_child procedure, 3-4
 DestroyNotify, B-4, B-7
 Destroy Callbacks, 2-23, 8-1
 destroy procedure, 2-24
 Display, 2-2
 display_accelerator procedure, 10-7, B-6

E

EastGravity, 2-19
 EnterNotify, 7-4, 7-10, B-2, B-4, B-7, B-8, B-9
 EnterWindow, 5-6
 Events, 7-7
 exit, 2-25
 expose procedure, 7-14
 Expose, 2-19, 7-13, 7-15, 9-19, 11-20, B-4, B-7

F

False, 1-8, 1-23, 2-17, 3-1, 3-7, 3-8, 4-7, 4-8, 4-9, 5-4, 5-6, 5-7, 7-5, 7-7, 7-10, 7-11, 7-14, 7-15, 11-7, 11-12, 11-14, 11-20
 FocusIn, 7-6, 7-7, 7-10, B-4, B-7
 FocusNotify, 7-6, 7-7
 FocusOut, 7-7, 7-10, B-4, B-7
 Foreground, 9-2
 free, 11-2

G

Geometry Management, 6-1
 geometry_manager procedure, 6-1
 get_values_hook procedure, 9-16
 Grabbing Input, 7-4
 GraphicsExpose, 7-16, 7-17, 7-18, 11-20, B-4, B-7

GravityNotify, B-4, B-7

H

hook, 9-16, 9-17

I

Inheritance, 1-21, 2-14, 2-16, 2-19, 9-6

Initialization, 1-22, 2-14, 2-16

initialize procedure, 2-14, 2-16

initialize_hook procedure, 2-16

Input Grabbing, 7-4

InputOnly, 2-20

InputOutput, 2-20

insert_child procedure, 1-25, 3-2, 3-3, 5-2, C-1

Instance, 1-2

K

key modifier, B-3

KeymapNotify, B-4, B-7

KeyPress, 7-4, 7-6, 7-10, B-2, B-4, B-5, B-7

KeyRelease, 7-4, 7-6, 7-10, B-2, B-4, B-7

L

LeaveNotify, 7-4, 7-10, B-2, B-4, B-7

libXt.a, 1-5

M

malloc, 11-2

MapNotify, B-4, B-7

MappingNotify, 7-16, 7-17, 7-18, B-4

MapRequest, B-4, B-7

MenuPopdown, 5-7, 5-8, 10-3, C-7

MenuPopup, 5-4, 5-5, 5-6, 10-3, C-7

Method, 1-3

MotionNotify, 7-4, 7-10, B-2, B-4, B-5, B-7

N

Name, 1-3

NoExpose, 7-16, 7-17, 7-18, B-4, B-7

None, 7-6

NorthWestGravity, 2-18, 7-14

O

Object, 1-3

Opposite, 6-4

OverrideShell, 4-1, 4-2, 4-7

OverrideShellWidget, 4-4

OverrideShellWidgetClass, 4-4

overrideShellWidgetClass, 4-4

OverrrideShell, 4-2

P

pop-up, 5-1

child, 5-1, 5-2

list, 5-1

shell, 5-2

printf, 11-23

PropertyNotify, B-4, B-7

Q

query_geometry procedure, 6-10

R

realize procedure, 2-19

realloc, 11-2

ReparentNotify, B-4, B-7

ResizeRequest, B-4, B-7

resize procedure, 6-11

Resource Management, 9-1

Resource, 1-3

S

SelectionClear, 7-16, 7-17, 7-18, B-4, B-7
SelectionNotify, 7-16, 7-17, 7-18, B-4, B-7
SelectionRequest, 7-16, 7-17, 7-18, B-4, B-7
selectionTimeout, 11-6
set_values procedure, 9-18, 9-21
set_values_almost procedure, 9-20
set_values_hook procedure, 9-22
Shell, 2-2, 4-1, 4-2, 4-7, 5-2, 5-4, 5-6, 5-7, 9-9
ShellPart, 4-5
ShellWidget, 4-4, 4-6
ShellWidgetClass, 4-4
shellWidgetClass, 4-4
String, 2-14
StringDefs.h, E-1
Subclass Chaining, 1-21
SubstructureNotify, 2-21
Superclass Chaining, 1-21, 2-14, 2-16, 9-6
superclass, 1-17

T

TARGETS, 11-8
TopIf, 6-4
TopLevel, 4-9
TopLevelShell, 4-1, 4-2
TopLevelShellWidget, 4-4
topLevelShellWidgetClass, 4-4, C-5
TopLevelShellWidgetClass, 4-5
TransientShell, 4-1, 4-2, 4-7, 4-8
TransientShellWidget, 4-4
transientShellWidgetClass, 4-4
TransientShellWidgetClass, 4-5
Translation Table, 10-4, B-1
True, 1-8, 1-20, 1-23, 2-4, 2-17, 2-18, 2-22, 3-1, 3-5, 3-7, 4-7, 4-8, 5-

4, 5-6, 6-3, 7-5, 7-6, 7-10, 7-11, 7-12, 7-13, 7-14, 7-15, 7-17, 9-18, 9-19, 9-20, 11-7, 11-12, 11-14, 11-20

U

UnmapNotify, B-4, B-7
User, 1-3

V

VendorShell, 4-2, 4-9
VendorShellWidget, 4-4
VendorShellWidgetClass, 4-4
vendorShellWidgetClass, 4-4
version, 1-17
Visibility, 7-15
VisibilityNotify, 7-15, B-4, B-7
Visible, 7-15

W

WestGravity, 2-19
Widget class, 1-3
Widget programmer, 1-3
Widget, 1-3, 1-7, 1-8
WidgetClass, 1-7, 1-17
widgetClass, 1-7
widgetClassRec, 1-17
WidgetClassRec, 1-7
WidgetList, 3-5
widget_class, 1-12
widget_size, 1-17
WMShell, 4-2, 4-8
WMShellWidget, 4-4
WMShellWidgetClass, 4-4
wmShellWidgetClass, 4-4

X

X11/Convert.h, 9-13, C-6
X11/Intrinsic.h, 1-4, 1-5
X11/IntrinsicP.h, 1-5
X11/keysymdef.h, B-6

X11/Label.h, 1-5
X11/Scroll.h, 1-5
X11/Shell.h, 1-4
X11/StringDefs.h, 1-4, 1-13, 9-2,
E-1
X11/X.h, 6-4
X11/Xatoms.h, 1-4
X11/Xresource.h, 9-9
X11/Xutil.h, 11-20
XA_CLIPBOARD, 11-11, 11-19
XA_PRIMARY, 11-7, 11-11, 11-14,
11-19
XA_SECONDARY, 11-7, 11-11,
11-14, 11-19
XA_STRING, 11-7, 11-9, 11-14,
11-16
XClearArea, 9-18, 9-20
XConfigureWindow, 2-21, 3-5, 6-3,
6-4, 6-8, 6-9
XCreateGC, 11-4, 11-5
XCreateWindow, 2-18, 2-19, 2-20
XDestroyWindow, 2-22, 2-23
XFreeGC, 2-24
XFreePixmap, 2-24
XMapWindow, 5-4, 5-6
xmh, 2-9
XMoveWindow, 3-5, 6-8
XNextEvent, 7-7
XOpenDisplay, 2-5, 2-6, 2-7
XPeekevent, 7-7
XPending, 7-7
XRInt, 9-9
XrmOptionDescRec, 2-7
XrmParseCommand, 2-4, 2-5, 2-7,
2-9, C-2
XrmValue, 9-10, 9-4, 9-9
XSelectInput, 7-16, 7-17, 7-18, 7-19
XSetInputFocus, 7-6, 7-7
XSetWindowAttributes, 2-17, 2-18,
2-19, 7-19
XSynchronize, 2-4
XtAcceptFocusProc, 7-7
XtActionList, 10-2
XtActionProc, 10-1
XtActionsRec, 10-2
XtAddActions, 10-3, C-7
XtAddCallback, 2-23, 8-2
XtAddCallbacks, 8-3
XtAddConverter, C-5
XtAddEventHandler, 2-24, 7-9, 7-
16, 7-17, 7-18, 7-19
XtAddExposureToRegion, 11-20
XtAddInput, C-1, C-3, C-4
XtAddGrab, 7-5, 7-10
XtAddRawEventHandler, 7-17, 7-18
XtAddress, 9-13, C-6
XtAddressMode, 9-13, C-6
XtAddTimeout, C-1, C-4
XtAddWorkProc, C-1, C-4
XtAllEvents, 7-17
XtAlmostProc, 9-20
XtAppAddActions, 10-2
XtAppAddConverter, 9-12
XtAppAddInput, 7-2, 7-3, C-4
XtAppAddTimeout, 2-24, 7-3, 7-4,
C-4
XtAppAddWorkProc, 7-12, C-5
XtAppContext, 2-2
XtAppCreateShell, 2-1, 2-2, 2-13,
2-14, 9-5
XtAppError, 11-26
XtAppErrorMsg, 11-24, 11-26
XtAppGetErrorDatabase, 11-22
XtAppGetErrorDatabaseText, 11-23
XtAppGetErrorDatabaseText, 11-22
XtAppGetSelectionTimeout, 11-6
XtAppMainLoop, 7-1, 7-9, 7-10, C-2
XtAppNextEvent, 7-8, 7-10, 7-11,
C-3
XtAppPeekEvent, 7-8, C-3
XtAppPending, 7-7, 7-8, C-3
XtAppProcessEvent, 7-8, 7-9, 7-11,
C-3
XtAppSetErrorHandler, 11-26
XtAppSetErrorMsgHandler, 11-23
XtAppSetSelectionTimeout, 11-6
XtAppSetWarningHandler, 11-26
XtAppSetWarningMsgHandler, 11-24

XtAppWarning, 11-26, 11-27
XtAppWarningMsg, 11-25, 11-27
XtArgsFunc, 9-22
XtArgsProc, 2-16, 9-16
XtArgVal, 2-10
XtAugmentTranslations, 10-5, 10-6
XtBaseOffset, 9-13, C-6
XtButtonBoxAddButton, C-1
XtButtonBoxDeleteButton, C-1
XtBuildEventMask, 7-19
XtC, 1-13, 9-2
XtCallAcceptFocus, 7-7
XtCallbackExclusive, 5-4, 5-5, 5-7
XtCallbackHasNone, 8-5
XtCallbackHasSome, 8-5
XtCallbackList, 8-1, 8-2
XtCallbackNoList, 8-5
XtCallbackNone, 5-4, 5-5, 5-7
XtCallbackNonexclusive, 5-4, 5-5, 5-7
XtCallbackPopdown, 5-7, 5-8
XtCallbackProc, 2-24, 8-1
XtCallbackRec, 8-2
XtCallCallbacks, 8-3, 8-4
XtCalloc, 2-24, 11-2, 11-3
XtCancelConvertSelectionProc, 11-15
XtCancelSelectionCallbackProc, 11-17
XtCaseProc, 10-10, 10-11
XtCheckSubclass, 1-20, 1-21, 5-4, 5-6, 5-7
XtClass, 1-20
XtCloseDisplay, 2-5, 2-6
XtConfigureWidget, 6-1, 6-2, 6-7, 6-8, 6-9
XtConvert, 9-14, 9-15
XtConvertArgRec, 9-13, C-6
XtConvertCase, 10-11
XtConverter, 9-9
XtConvertSelectionIncrProc, 11-13
XtConvertSelectionProc, 11-7, 11-8
XtCreateApplicationContext, 2-2, 2-3, C-2
XtCreateApplicationShell, C-1, C-5
XtCreateLabel, C-1
XtCreateManagedWidget, 2-13, 3-1, 3-5, 3-6
XtCreatePopupShell, 2-14, 5-2
XtCreateWidget, 1-8, 1-23, 2-9, 2-11, 2-12, 2-17, 3-1, 3-2, 3-4, 3-5, 3-6, 3-9, 8-2, 9-1, 9-6, 9-16, 9-22, C-1
XtCreateWindow, 2-19, 2-20, 2-21
XtCWQueryOnly, 6-3, 6-4, 6-5, 6-6, 6-7
XtDatabase, 2-7
XtDefaultBackground, 1-8, 2-4, 2-9, 9-9, 9-12
XtDefaultFont, 9-9, 9-12
XtDefaultForeground, 1-8, 2-4, 2-9, 9-3, 9-9, 9-12
XtDestroyApplicationContext, 2-3, 2-6, 2-25
XtDestroyGC, 2-24, C-7
XtDestroyWidget, 2-1, 2-17, 2-22, 2-23, 2-24, 3-1, 3-4, 3-10, 5-1
XtDirectConvert, 9-14, 9-15
XtDisownSelection, 11-12, 11-20
XtDispatchEvent, 2-23, 7-5, 7-6, 7-9, 7-10, C-2
XtDisplay, 2-20
XtDisplayInitialize, 2-2, 2-3, 2-4, 2-5, 2-6, 2-7, 2-9, 2-13, C-2
XtError, 11-24, C-9, C-11
XtErrorHandler, 11-25
XtErrorMsg, 1-21, 11-2, 11-3, 11-4, C-9, C-11
XtErrorMsgHandler, 11-22
XtEventHandler, 7-15
XtExposeProc, 7-14
XtFree, 2-11, 2-24, 9-4, 11-2, 11-3, 11-7, 11-9, 11-16
XtGeometryAlmost, 6-2, 6-5, 6-6, 6-7, 6-10, 6-11, 9-18, 9-20
XtGeometryDone, 6-3, 6-6
XtGeometryHandler, 6-6, 6-10
XtGeometryMask, 6-3
XtGeometryNo, 4-8, 6-3, 6-7, 6-11,

9-18
 XtGeometryResult, 6-3
 XtGeometryYes, 6-2, 6-3, 6-6, 6-7, 6-10, 6-11, 9-18
 XtGetApplicationResources, 9-14, 9-7, 9-8
 XtGetErrorDatabase, C-8
 XtGetErrorDatabaseText, C-8, C-9
 XtGetErrorDatabaseText, C-8
 XtGetGC, 2-24, 11-4, 11-5
 XtGetResourceList, 9-4, 9-5
 XtGetSelectionTimeout, C-8
 XtGetSelectionValue, 11-9, 11-10, 11-11, 11-18
 XtGetSelectionValueIncremental, 11-17, 11-18, 11-19
 XtGetSelectionValues, 11-9, 11-10, 11-11
 XtGetSelectionValuesIncremental, 11-17, 11-18, 11-19
 XtGetSubresources, 9-6, 9-7, 9-14
 XtGetSubvalues, 9-16, 9-17
 XtGetValues, 3-9, 8-1, 8-2, 9-1, 9-5, 9-15, 9-16
 XtGrabExclusive, 5-4, 5-5, 5-6, 5-7
 XtGrabKind, 5-4
 XtGrabNone, 5-5
 XtGrabNonexclusive, 5-4, 5-5, 5-6, 5-7
 XtHasCallbacks, 8-5
 XtIMAll, 7-9
 XtIMAlternateInput, 7-8, 7-9
 XtImmediate, 9-13, C-6
 XtIMTimer, 7-8, 7-9
 XtInherit, 1-24
 XtInheritAcceptFocus, 1-25
 XtInheritChangeManaged, 1-25
 XtInheritDeleteChild, 1-25
 XtInheritDisplayAccelerator, 1-25
 XtInheritExpose, 1-25
 XtInheritGeometryManager, 1-25
 XtInheritInsertChild, 1-25
 XtInheritRealize, 1-25
 XtInheritResize, 1-25
 XtInheritSetValuesAlmost, 1-25, 9-20
 XtInheritTranslations, 10-4
 XtInitialize, C-1, C-2, C-3, C-4, C-5
 XtInitProc, 2-14, 2-16
 XtInputCallbackProc, 7-2
 XtInputExceptMask, 7-2
 XtInputReadMask, 7-2
 XtInputWriteMask, 7-2
 XtInstallAccelerators, 10-8
 XtInstallAllAccelerators, 10-8
 XtIsComposite, 3-2
 XtIsManaged, 3-7
 XtIsRealized, 2-17, 2-18
 XtIsSensitive, 7-11
 XtIsSubclass, 1-20, 3-2
 XtKeyProc, 10-9, 10-11, B-4
 XtLabelCreate, C-1
 XtLoseSelectionIncrProc, 11-14
 XtLoseSelectionProc, 11-8
 XtMainLoop, C-1, C-2
 XtMakeGeometryRequest, 2-1, 6-1, 6-2, 6-3, 6-4, 6-5, 6-6, 6-7, 6-12
 XtMakeResizeRequest, 6-1, 6-5, 6-12
 XtMalloc, 2-24, 11-2, 11-3, 11-4
 XtManageChild, 1-25, 2-9, 3-1, 3-5, 3-6, C-1
 XtManageChildren, 2-17, 3-1, 3-4, 3-5, C-1
 XtMapWidget, 3-8
 XtMergeArgLists, 2-11
 XtMoveWidget, 3-5, 6-1, 6-2, 6-7, 6-8
 XtN, 1-13, 9-2
 XtNameToWidget, 11-1, 11-2
 XtNew, 11-3, 11-4
 XtNewString, 11-4
 XtNextEvent, C-1, C-2, C-3
 XtNumber, 2-11, 2-13, 11-1
 XtOffset, 9-3, 9-5, 9-6
 XtOpenDisplay, 2-2, 2-4, 2-5, 2-7
 XtOrderProc, 3-3
 XtOverrideTranslations, 10-5, 10-6

XtOwnSelection, 11-11, 11-12
XtOwnSelectionIncremental, 11-14, 11-19, 11-20
XtParent, 2-20, 2-21
XtParseAcceleratorTable, 10-8
XtParseTranslationTable, 10-4, 10-5
XtPeekEvent, C-1, C-3
XtPending, C-1, C-3
XtPopdown, 4-8, 5-6, 5-7, 5-8
XtPopdownID, 5-7
XtPopup, 4-8, 5-3, 5-4, 5-5, 5-6, 7-4, 7-5
XtProc, 1-22
XtProcessEvent, C-1, C-3
XtQueryGeometry, 6-9, 6-10, 6-11
XtR, 1-13
XtRAcceleratorTable, 9-3, 9-9
XTranslateCoordinates, 11-21
XtReleaseGC, C-7
XtRBool, 9-3, 9-9
XtRBoolean, 9-3, 9-9
XtRCallback, 8-2, 9-3
XtRCallProc, 9-4
XtRColor, 9-3, 9-9
XtRCursor, 9-3, 9-9
XtRDimension, 9-3, 9-9
XtRDisplay, 9-3, 9-9
XtRealizeProc, 2-18
XtRealizeWidget, 2-1, 2-9, 2-10, 2-16, 2-17, 2-18, 2-19, 2-22, 3-5, 5-3, 5-4, 5-6, 7-14, 7-19
XtRealloc, 11-2, 11-3
XtRegisterCaseConverter, 10-10, 10-11
XtReleaseGC, 11-5
XtRemoveAllCallbacks, 8-4
XtRemoveCallback, 2-24, 8-3, 8-4
XtRemoveCallbacks, 8-4
XtRemoveEventHandler, 2-24, 7-17
XtRemoveGrab, 5-7, 7-5, 7-6
XtRemoveInput, 7-3
XtRemoveRawEventHandler, 7-18, 7-19
XtRemoveTimeOut, 2-24, 7-4
XtRemoveWorkProc, 7-12
XtResizeWidget, 3-5, 6-11, 6-1, 6-2, 6-7, 6-8, 6-9
XtResizeWindow, 6-9
XtResource, 9-1
XtResourceDefaultProc, 9-4
XtResourceList, 1-12, 9-1
XtResourceQuark, 9-13, C-6
XtResourceString, 9-13, C-6
XtRFile, 9-3, 9-9
XtRFloat, 9-3, 9-9
XtRFont, 9-3, 9-9
XtRFontStruct, 9-3, 9-9
XtRFunction, 9-3
XtRImmediate, 9-4
XtRInt, 9-3, 9-9
XtRPixel, 9-3, 9-9
XtRPixmap, 9-3, 9-9
XtRPointer, 9-3
XtRPosition, 9-3, 9-9
XtRShort, 9-3, 9-9
XtRString, 9-3, 9-9
XtRTranslationTable, 9-3, 9-9
XtRUnsignedChar, 9-3, 9-9
XtRWidget, 9-3
XtRWindow, 9-3
XtScreen, 2-21
XtSelectionCallbackProc, 11-8
XtSelectionDoneIncrProc, 11-15
XtSelectionDoneProc, 11-7, 11-8
XtSelectionIncrCallbackProc, 11-16
XtSetArg, 2-10, 2-11
XtSetErrorHandler, C-10
XtSetErrorMsgHandler, C-9
XtSetKeyboardFocus, 7-6
XtSetKeyTranslator, 10-9
XtSetMappedWhenManaged, 3-1, 3-7, 3-8
XtSetSelectionTimeout, C-7
XtSetSensitive, 5-3, 5-5, 5-8, 7-11
XtSetSubvalues, 9-21
XtSetValues, 1-13, 2-21, 3-8, 3-9, 5-3, 6-1, 6-2, 7-11, 8-2, 9-1, 9-5, 9-17, 9-18, 9-20, 9-22, 10-6

XtSetValuesFunc, **9-18**, 9-21
XtSetWarningHandler, **C-11**
XtSetWarningMsgHandler, **C-9**
XtSMDontChange, 6-4, 6-11
XtStringConversionWarning, **9-12**
XtStringProc, **10-7**
XtSuperclass, **1-20**
XtTimerCallbackProc, **7-4**
XtToolkitError, 11-24, 11-25, C-9,
C-10, D-1
XtToolkitInitialize, **2-2**, C-2
XtTranslateCoords, **11-21**
XtTranslateKey, 10-9
XtTranslateKeycode, **10-9**, 10-10
XtTranslations, 10-5
XtUninstallTranslations, **10-6**, 10-7
XtUnmanageChild, 2-23, 3-1, **3-6**,
3-7
XtUnmanageChildren, 2-17, 3-1, **3-6**,
3-7, C-1
XtUnmapWidget, 2-25, **3-8**
XtUnrealizeWidget, **2-21**, 2-22
XtVersion, 1-17
XtVersionDontCheck, 1-17
XtWarning, 11-25, C-10, **C-11**
XtWarningMsg, 9-10, **C-10**, C-11
XtWidgetClassProc, **1-23**
XtWidgetGeometry, 6-3, 6-4, 6-5,
6-10
XtWidgetProc, **2-24**, 2-25, 3-3, 3-4,
6-11
XtWidgetToApplicationContext, **2-3**
XtWindow, **2-21**
XtWindowToWidget, **11-21**
XtWorkProc, **7-12**
XtWorkProcId, 7-12
XT_CONVERT_FAIL, 11-9, 11-16

—

XtError, 11-26, C-10
XtInherit, 1-24
XtWarning, 11-26, C-11

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA
and New Hampshire,
Alaska or Hawaii
call **800-DIGITAL**

In Canada
call **800-267-6215**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminister, Massachusetts 01473

*Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
809-754-7575

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. _____

Did you find errors in this manual? If so, specify the error and the page number. _____

Please indicate the type of user/reader that you most nearly represent.

- Assembly language programmer
- Higher-level language programmer
- Occasional programmer (experienced)
- User with little programming experience
- Student programmer
- Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____

or _____
Country _____

Do Not Tear - Fold Here and Tape

digital™



No Postage
Necessary
if Mailed
in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Documentation Manager
ULTRIX Documentation Group
ZKO3-3/X18
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987



Do Not Tear - Fold Here

Cut Along Dotted Line