X Toolkit Intrinsics – C Language Interface

X Window System

X Version 11, Release 5

First Revision - August, 1991

Joel McCormack

Digital Equipment Corporation
Western Software Laboratory

Paul Asente

Digital Equipment Corporation
Western Software Laboratory

Ralph R. Swick

Digital Equipment Corporation
External Research Group
MIT X Consortium

Acknowledgments

The design of the X11 Intrinsics was done primarily by Joel McCormack of Digital WSL. Major contributions to the design and implementation also were done by Charles Haynes, Mike Chow, and Paul Asente of Digital WSL. Additional contributors to the design and/or implementation were:

Loretta Guarino-Reid (Digital WSL)
Rich Hyde (Digital WSL)
Susan Angebranndt (Digital WSL)
Terry Weissman (Digital WSL)
Mary Larson (Digital UEG)
Mark Manasse (Digital SRC)
Jim Gettys (Digital SRC)
Ralph Swick (Project Athena and Digital ERP)
Leo Treggiari (Digital SDT)
Ron Newman (Project Athena)
Mark Ackerman (Project Athena)
Bob Scheifler (MIT LCS)

The contributors to the X10 toolkit also deserve mention. Although the X11 Intrinsics present an entirely different programming style, they borrow heavily from the implicit and explicit concepts in the X10 toolkit.

The design and implementation of the X10 Intrinsics were done by:

Terry Weissman (Digital WSL)
Smokey Wallace (Digital WSL)
Phil Karlton (Digital WSL)
Charles Haynes (Digital WSL)
Frank Hall (HP)

The design and implementation of the X10 toolkit's sample widgets were by the above, as well as by:

Ram Rao (Digital UEG)
Mary Larson (Digital UEG)
Mike Gancarz (Digital UEG)
Kathleen Langone (Digital UEG)

These widgets provided a checklist of requirements that we had to address in the X11 intrinsics.

Thanks go to Al Mento of Digital's UEG Documentation Group for formatting and generally improving this document and to John Ousterhout of Berkeley for extensively reviewing early drafts of it.

Finally, a special thanks to Mike Chow, whose extensive performance analysis of the X10 toolkit provided the justification to redesign it entirely for X11.

Joel McCormack
Western Software Laboratory
Digital Equipment Corporation

March, 1988

The current design of the Intrinsics has benefited greatly from the input of several dedicated reviewers in the membership of the X Consortium. In addition to those already mentioned, the following individuals have dedicated significant time to suggesting improvements to the Intrinsics:

Steve Pitschke (Stellar)	C. Doug Blewett (AT&T)
Bob Miller (HP)	David Schiferl (Tektronix)
Fred Taft (HP)	Michael Squires (Sequent)
Marcel Meth (AT&T)	Jim Fulton (MIT)
Mike Collins (Digital)	Kerry Kimbrough (Texas Instruments)
Scott McGregor (Digital)	Phil Karlton (Digital)
Julian Payne (ESS)	Jacques Davy (Bull)
Gabriel Beged-Dov (HP)	Glenn Widener (Tektronix)

Thanks go to each of them for the countless hours spent reviewing drafts and code.

Ralph R. Swick
External Research Group
Digital Equipment Corporation
MIT Project Athena

June, 1988

From Release 3 to Release 4, several new members joined the design team. We greatly appreciate the thoughtful comments, suggestions, lengthy discussions, and in some cases implementation code contributed by each of the following:

Don Alecci (AT&T)	Ellis Cohen (OSF)
Donna Converse (MIT)	Clive Feather (IXI)
Nayeem Islam (Sun)	Dana Laursen (HP)
Keith Packard (MIT)	Chris Peterson (MIT)
Richard Probst (Sun)	Larry Cable (Sun)

In Release 5, the effort to define the internationalization additions was headed by Bill McMahon of Hewlett Packard and Frank Rojas of IBM. This has been an educational process for many of us and Bill and Frank's tutelage has carried us through. Vania Joloboff of the OSF also contributed to the internationalization additions. The implementation efforts of Bill, Gabe Beged-Dov, and especially Donna Converse for this release are also gratefully acknowledged.

Ralph R. Swick

December, 1989
and
July, 1991

About This Manual

X Toolkit Intrinsics – C Language Interface is intended to be read by both application programmers who will use one or more of the many widget sets built with the Intrinsics and by widget programmers who will use the Intrinsics to build widgets for one of the widget sets. Not all the information in this manual, however, applies to both audiences. That is, because the application programmer is likely to use only a number of the Intrinsics functions in writing an application and because the widget programmer is is likely to use many more, if not all, of the Intrinsics functions in building a widget, an attempt has been made to highlight those areas of information that are deemed to be of special interest for the application programmer. (It is assumed the widget programmer will have to be familiar with all the information.) Therefore, all entries in the table of contents that are printed in **bold** indicate the information that should be of special interest to an application programmer.

It is also assumed that as application programmers become more familiar with the concepts discussed in this manual they will find it more convenient to implement portions of their applications as special-purpose or custom widgets. It is possible, none the less, to use widgets without knowing how to build them.

Conventions Used in this Manual

This document uses the following conventions:

- Global symbols are printed in **this special font**. These can be either function names, symbols defined in include files, data types, or structure names. Arguments to functions, procedures, or macros are printed in *italics*.
- 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.
- 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*.

Chapter 1

Intrinsics and Widgets

The Intrinsics are a programming library tailored to the special requirements of user interface construction within a network window system, specifically the X Window System. The Intrinsics and a widget set make up an X Toolkit.

1.1. Intrinsics

The Intrinsics provide the base mechanism necessary to build a wide variety of interoperating widget sets and application environments. The Intrinsics are a layer on top of Xlib, the C Library X Interface. They extend the fundamental abstractions provided by the X Window System while still remaining independent of any particular user interface policy or style.

The Intrinsics use object-oriented programming techniques to supply a consistent architecture for constructing and composing user interface components, known as widgets. This allows programmers to extend a widget set in new ways, either by deriving new widgets from existing ones (subclassing), or by writing entirely new widgets following the established conventions.

When the Intrinsics were first conceived, the root of the object hierarchy was a widget class named Core. In release 4 of the Intrinsics, three nonwidget superclasses were added above Core. These superclasses are described in Chapter 12. The name of the class now at the root of the Intrinsics class hierarchy is Object. The remainder of this specification refers uniformly to *widgets* and *Core* as if they were the base class for all Intrinsics operations. The argument descriptions for each Intrinsica procedure and Chapter 12 describe which operations are defined for the nonwidget superclasses of Core. The reader may determine by context whether a specific reference to *widget* actually means *widget or object*.

1.2. Languages

The Intrinsics are intended to be used for two programming purposes. Programmers writing widgets will be using most of the facilities provided by the Intrinsics to construct user interface components from the simple, such as buttons and scrollbars, to the complex, such as control panels and property sheets. Application programmers will use a much smaller subset of the Intrinsics procedures in combination with one or more sets of widgets to construct and present complete user interfaces on an X display. The Intrinsics programming interfaces primarily intended for application use are designed to be callable from most procedural programming languages. Therefore, most arguments are passed by reference rather than by value. The interfaces primarily intended for widget programmers are expected to be used principally from the C language. In these cases, the usual C programming conventions apply. In this specification, the term *client* refers to any module, widget, or application that calls an Intrinsics procedure.

Applications that use the Intrinsics mechanisms must include the header files <**X11/Intrinsic.h**> and <**X11/StringDefs.h**>, or their equivalent, and they may also include <**X11/Xatoms.h**> and <**X11/Shell.h**>. In addition, widget implementations should include <**X11/IntrinsicP.h**> instead of <**X11/Intrinsic.h**>.

The applications must also include the additional header files for each widget class that they are to use (for example, <**X11/Xaw/Label.h**> or <**X11/Xaw/Scrollbar.h**>). On a POSIX-based system, the Intrinsics object library file is named **libXt.a** and is usually referenced as –lXt when linking the application.

1.3. Procedures and Macros

All functions defined in this specification except those specified below may be implemented as C macros with arguments. C applications may use ‘‘#undef’’ to remove a macro definition and ensure that the actual function is referenced. Any such macro will expand to a single expression which has the same precedence as a function call and that evaluates each of its arguments exactly once, fully protected by parentheses, so that arbitrary expressions may be used as arguments.

The following symbols are macros that do not have function equivalents and that may expand their arguments in a manner other than that described above: **XtCheckSubclass**, **XtNew**, **XtNumber**, **XtOffsetOf**, **XtOffset**, and **XtSetArg**.

1.4. Widgets

The fundamental abstraction and data type of the X Toolkit is the widget, which is a combination of an X window and its associated input and display semantics and which is dynamically allocated and contains state information. 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, which is statically allocated and initialized and which contains the operations allowable on widgets of that class. Logically, a widget class is the procedures and data 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 2.5.

The distribution of the declarations and code for a new widget class among a public .h file for application programmer use, a private .h file for widget programmer use, and the implementation .c file is described in Section 1.6. The predefined widget classes adhere to these conventions.

A widget instance is composed of two parts:

- A data structure which contains instance-specific values.
- A class structure which 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.

This chapter discusses the base widget classes, Core, Composite, and Constraint, and ends with a discussion of widget classing.

1.4.1. Core Widgets

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

1.4.1.1. CoreClassPart Structure

All widget classes contain the fields defined in the **CoreClassPart** structure.

```
typedef struct {
	WidgetClass superclass;			See Section 1.6
	String class_name;			See Chapter 9
	Cardinal widget_size;			See Section 1.6
	XtProc class_initialize;		See Section 1.6
	XtWidgetClassProc class_part_initialize;See Section 1.6
	XtEnum class_inited;			See Section 1.6
	XtInitProc initialize;			See Section 2.5
	XtArgsProc initialize_hook;		See Section 2.5
	XtRealizeProc realize;			See Section 2.6
	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
	XtEnum compress_exposure;		See Section 7.9
	Boolean compress_enterleave;		See Section 7.9
	Boolean visible_interest;		See Section 7.10
	XtWidgetProc destroy;			See Section 2.8
	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 1.6
	XtPointer callback_private;		Private to callbacks
	String tm_table;			See Chapter 10
	XtGeometryHandler query_geometry;	See Chapter 6
	XtStringProc display_accelerator;	See Chapter 10
	XtPointer extension;			See Section 1.6
} CoreClassPart;
```

All widget classes have the Core class fields as their first component. The prototypical **WidgetClass** and **CoreWidgetClass** are defined with only this set of fields.

```
typedef struct {
	CoreClassPart core_class;
} WidgetClassRec, *WidgetClass, CoreClassRec, *CoreWidgetClass;
```

Various routines can cast widget class pointers, as needed, to specific widget class types.

The single occurrences of the class record and pointer for creating instances of Core are

In **IntrinsicP.h**:

```
extern WidgetClassRec widgetClassRec;
#define coreClassRec widgetClassRec
```

In **Intrinsic.h**:

```
extern WidgetClass widgetClass, coreWidgetClass;
```

The opaque types **Widget** and **WidgetClass** and the opaque variable **widgetClass** are defined for generic actions on widgets. In order to make these types opaque and ensure that the compiler does not allow applications to access private data, the Intrinsics use incomplete structure definitions in **Intrinsic.h**:

```
typedef struct _WidgetClassRec *WidgetClass, *CoreWidgetClass;
```

1.4.1.2. CorePart Structure

All widget instances contain the fields defined in the **CorePart** structure.

```
typedef struct _CorePart {
	Widget self;	described below
	WidgetClass widget_class;	See Section 1.6
	Widget parent;	See Section 2.5
	Boolean being_destroyed;	See Section 2.8
	XtCallbackList destroy_callbacks;	See Section 2.8
	XtPointer constraints;	See Section 3.6
	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
	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 Setcion 2.6
	Window window;	See Section 2.6
	Cardinal depth;	See Section 2.6
	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.

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

Various routines can cast widget pointers, as needed, to specific widget types.

In order to make these types opaque and ensure that the compiler does not allow applications to access private data, the Intrinsics use incomplete structure definitions in **Intrinsic.h**.

```
typedef struct _WidgetRec *Widget, *CoreWidget;
```

1.4.1.3. Core Resources

The resource names, classes, and representation types specified in the **coreClassRec** resource list are

Name	Class	Representation
XtNaccelerators	XtCAccelerators	XtRAcceleratorTable
XtNbackground	XtCBackground	XtRPixel
XtNbackgroundPixmap	XtCPixmap	XtRPixmap
XtNborderColor	XtCBorderColor	XtRPixel
XtNborderPixmap	XtCPixmap	XtRPixmap
XtNcolormap	XtCColormap	XtRColormap
XtNdepth	XtCDepth	XtRInt
XtNmappedWhenManaged	XtCMappedWhenManaged	XtRBoolean
XtNscreen	XtCScreen	XtRScreen
XtNtranslations	XtCTranslations	XtRTranslationTable

Additional resources are defined for all widgets via the **objectClassRec** and **rectObjClassRec** resource lists; see Sections 12.2 and 12.3 for details.

1.4.1.4. CorePart Default Values

The default values for the Core fields, which are filled in from the resource lists and by the initialize procedures, are

Field	Default Value
self	Address of the widget structure (may not be changed).
widget_class	*widget_class* argument to **XtCreateWidget** (may not be changed).
parent	*parent* argument to **XtCreateWidget** (may not be changed).
being_destroyed	Parent's *being_destroyed* value.
destroy_callbacks	NULL
constraints	NULL
x	0
y	0
width	0
height	0
border_width	1
managed	**False**
sensitive	**True**
ancestor_sensitive	logical AND of parent's *sensitive* and *ancestor_sensitive* values.
accelerators	NULL
border_pixel	**XtDefaultForeground**
border_pixmap	**XtUnspecifiedPixmap**
popup_list	NULL
num_popups	0
name	*name* argument to **XtCreateWidget** (may not be changed).
screen	Parent's *screen*; top-level widget gets screen from display specifier (may not be changed).
colormap	Parent's *colormap* value.
window	NULL
depth	Parent's *depth*; top-level widget gets root window depth.

background_pixel	**XtDefaultBackground**
background_pixmap	**XtUnspecifiedPixmap**
visible	**True**
mapped_when_managed	**True**

XtUnspecifiedPixmap is a symbolic constant guaranteed to be unequal to any valid Pixmap id, **None**, and **ParentRelative**.

1.4.2. Composite Widgets

The Composite widget class is a subclass of the Core widget class (see Chapter 3). Composite widgets are intended to be containers for other widgets. The additional data used by composite widgets are defined by the **CompositeClassPart** and **CompositePart** structures.

1.4.2.1. CompositeClassPart Structure

In addition to the Core class fields, widgets of the Composite class 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
	XtPointer extension;			See Section 1.6
} CompositeClassPart;
```

The extension record defined for **CompositeClassPart** with *record_type* equal to **NULLQUARK** is **CompositeClassExtensionRec**.

```
typedef struct {
	XtPointer next_extension;		See Section 1.6.12
	XrmQuark record_type;			See Section 1.6.12
	long version;				See Section 1.6.12
	Cardinal record_size;			See Section 1.6.12
	Boolean accepts_objects;		See Chapter 3
} CompositeClassExtensionRec, *CompositeClassExtension;
```

Composite classes have the Composite class fields immediately following the Core class fields.

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

The single occurrences of the class record and pointer for creating instances of Composite are

In **IntrinsicP.h**:

extern CompositeClassRec compositeClassRec;

In **Intrinsic.h**:

extern WidgetClass compositeWidgetClass;

The opaque types **CompositeWidget** and **CompositeWidgetClass** and the opaque variable **compositeWidgetClass** are defined for generic operations on widgets whose class is Composite or a subclass of Composite. The symbolic constant for the **CompositeClassExtension** version identifier is **XtCompositeExtensionVersion** (see Section 1.6.12). **Intrinsic.h** uses an incomplete structure definition to ensure that the compiler catches attempts to access private data.

```
typedef struct _CompositeClassRec *CompositeWidgetClass;
```

1.4.2.2. CompositePart Structure

In addition to the Core instance fields, widgets of the Composite class have the following instance fields defined in the **CompositePart** structure.

```
typedef struct {
	WidgetList children;			See Chapter 3
	Cardinal num_children;			See Chapter 3
	Cardinal num_slots;			See Chapter 3
	XtOrderProc insert_position;		See Section 3.2
} CompositePart;
```

Composite widgets have the Composite instance fields immediately following the Core instance fields.

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

Intrinsic.h uses an incomplete structure definition to ensure that the compiler catches attempts to access private data.

```
typedef struct _CompositeRec *CompositeWidget;
```

1.4.2.3. Composite Resources

The resource names, classes, and representation types that are specified in the **compositeClassRec** resource list are

Name	Class	Representation
XtNchildren	XtCReadOnly	XtRWidgetList
XtNinsertPosition	XtCInsertPosition	XtRFunction
XtNnumChildren	XtCReadOnly	XtRCardinal

1.4.2.4. CompositePart Default Values

The default values for the Composite fields, which are filled in from the Composite resource list and by the Composite initialize procedure, are

Field	Default Value
children	NULL
num_children	0
num_slots	0
insert_position	Internal function to insert at end

The *children*, *num_children*, and *insert_position* fields are declared as resources; XtNinsertPosition is a settable resource, XtNchildren and XtNnumChildren may be read by any client but should only be modified by the composite widget class procedures.

1.4.3. Constraint Widgets

The Constraint widget class is a subclass of the Composite widget class (see Section 3.6). Constraint widgets maintain additional state data for each child; for example, client-defined constraints on the child's geometry. The additional data used by constraint widgets are defined by the **ConstraintClassPart** and **ConstraintPart** structures.

1.4.3.1. ConstraintClassPart Structure

In addition to the Core and Composite class fields, widgets of the Constraint class have the following class fields.

```
typedef struct {
	XtResourceList resources;	See Chapter 9
	Cardinal num_resources;	See Chapter 9
	Cardinal constraint_size;	See Section 3.6
	XtInitProc initialize;	See Section 3.6
	XtWidgetProc destroy;	See Section 3.6
	XtSetValuesFunc set_values;	See Section 9.7.2
	XtPointer extension;	See Section 1.6
} ConstraintClassPart;
```

The extension record defined for **ConstraintClassPart** with *record_type* equal to **NULLQUARK** is **ConstraintClassExtensionRec**.

```
typedef struct {
	XtPointer next_extension;	See Section 1.6.12
	XrmQuark record_type;	See Section 1.6.12
	long version;	See Section 1.6.12
	Cardinal record_size;	See Section 1.6.12
	XtArgsProc get_values_hook;	See Section 9.7.1
} ConstraintClassExtensionRec, *ConstraintClassExtension;
```

Constraint classes have the Constraint class fields immediately following the Composite class fields.

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

The single occurrences of the class record and pointer for creating instances of Constraint are

In **IntrinsicP.h**:

```
extern ConstraintClassRec constraintClassRec;
```

In **Intrinsic.h**:

```
extern WidgetClass constraintWidgetClass;
```

The opaque types **ConstraintWidget** and **ConstraintWidgetClass** and the opaque variable **constraintWidgetClass** are defined for generic operations on widgets whose class is Constraint or a subclass of Constraint. The symbolic constant for the **ConstraintClassExtension** version identifier is **XtConstraintExtensionVersion** (see Section 1.6.12). **Intrinsic.h** uses an incomplete structure definition to ensure that the compiler catches attempts to access private data.

```
typedef struct _ConstraintClassRec *ConstraintWidgetClass;
```

1.4.3.2. ConstraintPart Structure

In addition to the Core and Composite instance fields, widgets of the Constraint class have the following unused instance fields defined in the **ConstraintPart** structure

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

Constraint widgets have the Constraint instance fields immediately following the Composite instance fields.

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

Intrinsic.h uses an incomplete structure definition to ensure that the compiler catches attempts to access private data.

```
typedef struct _ConstraintRec *ConstraintWidget;
```

1.4.3.3. Constraint Resources

The **constraintClassRec** *core_class* and *constraint_class resources* fields are NULL and the *num_resources* fields are zero; no additional resources beyond those declared by the superclasses are defined for Constraint.

1.5. Implementation-specific Types

To increase the portability of widget and application source code between different system environments, the Intrinsics define several types whose precise representation is explicitly dependent upon, and chosen by, each individual implementation of the Intrinsics.

These implementation-defined types are

Boolean A datum that contains a zero or nonzero value. Unless explicitly stated, clients should not assume that the nonzero value is equal to the symbolic value **True**.

Cardinal An unsigned integer datum with a minimum range of [0..2^16-1]

Dimension An unsigned integer datum with a minimum range of [0..2^16-1]

Position A signed integer datum with a minimum range of [-2^15..2^15-1]

XtPointer A datum large enough to contain the largest of a char*, int*, function pointer, structure pointer, or long value. A pointer to any type or function, or a long value may be converted to an **XtPointer** and back again and the result will compare equal to the original value. In ANSI C environments it is expected that **XtPointer** will be defined as void*.

XtArgVal A datum large enough to contain an **XtPointer**, **Cardinal**, **Dimension**, or **Position** value.

XtEnum An integer datum large enough to encode at least 128 distinct values, two of which are the symbolic values **True** and **False**. The symbolic values **TRUE** and **FALSE** are also defined to be equal to **True** and **False**, respectively.

In addition to these specific types, the precise order of the fields within the structure declarations for any of the instance part records **ObjectPart**, **RectObjPart**, **CorePart**, **CompositePart**, **ShellPart**, **WMShellPart**, **TopLevelShellPart**, and **ApplicationShellPart** is implementation-defined. These structures may also have additional private fields internal to the implementation. The **ObjectPart**, **RectObjPart**, and **CorePart** structures must be defined so that any member with the same name appears at the same offset in **ObjectRec**, **RectObjRec** and **CoreRec** (**WidgetRec**). No other relations between the offsets of any two fields may be assumed.

1.6. 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, widgets usually do not implement directly callable procedures; rather, they implement procedures, called methods, that are available through their widget class structure. These methods are invoked by generic procedures that envelop common actions around the methods implemented by the widget class. Such procedures are applicable to all widgets of that class and also to widgets whose classes 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 necessary 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 superclasses' 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 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, used by client widgets or applications.
- A private .h file, used by widgets whose classes are subclasses of the widget class.
- A .c file, which implements the widget.

1.6.1. 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 library naming conventions that are applicable. For example, a record component name is all lower case and uses underscores (_) for compound words (for example, background_pixmap). Type and procedure names start with upper case and use capitalization for compound words (for example, **ArgList** or **XtSetValues**).
- A resource name 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 symbolic identifier prefixed with ‘‘XtN’’. For example, the *background_pixmap* field has the corresponding 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 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 symbolic

identifier prefixed with ‘‘XtC’’ (for example, XtCBorderWidth). Many predefined classes are listed in <**X11/StringDefs.h**>.

- A resource representation string is spelled identically to the type name (for example, ‘‘TranslationTable’’). Each representation string should have a symbolic identifier prefixed with ‘‘XtR’’ (for example, XtRTranslationTable). Many predefined representation types are listed in <**X11/StringDefs.h**>.
- New widget classes start with a capital and use upper case for compound words. Given a new class name AbcXyz, you should derive several names:
  – Additional widget instance structure part name AbcXyzPart.
  – Complete widget instance structure names AbcXyzRec and _AbcXyzRec.
  – Widget instance structure pointer type name AbcXyzWidget.
  – Additional class structure part name AbcXyzClassPart.
  – Complete class structure names AbcXyzClassRec and _AbcXyzClassRec.
  – Class structure pointer type name AbcXyzWidgetClass.
  – Class structure variable abcXyzClassRec.
  – Class structure 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 upper case (for example, ‘‘Highlight’’ and ‘‘NotifyClient’’).

The symbolic identifiers XtN..., XtC... and XtR... may be implemented as macros, as global symbols, or as a mixture of the two. The (implicit) type of the identifier is **String**. The pointer value itself is not significant; clients must not assume that inequality of two identifiers implies inequality of the resource name, class, or representation string. Clients should also note that although global symbols permit savings in literal storage in some environments, they also introduce the possibility of multiple definition conflicts when applications attempt to use independently developed widgets simultaneously.

1.6.2. 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 file for the superclass.
- Symbolic identifiers for the names and classes of the new resources that this widget adds to its superclass. The definitions should have a single space between the definition name and the value and no trailing space or comment in order to reduce the possibility of compiler warnings from similar declarations in multiple classes.
- Type declarations for any new resource data types defined by the class.
- The class record pointer variable used to create widget instances.
- The C type that corresponds to 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 LabelSetText();
        /* Widget w */
        /* String text */

extern String LabelGetText();
        /* 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 class is **Constraint.h**.

1.6.3. 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.
- Symbolic identifiers for any new resource representation types defined by the class. The definitions should have a single space between the definition name and the value and no trailing space or comment.
- A structure part definition for the new fields that the widget instance adds to its superclass's widget structure.
- The complete widget instance structure definition for this widget.
- A structure part definition for the new fields that this widget class adds to its superclass's constraint structure if the widget class is a subclass of Constraint.
- The complete constraint structure definition if the widget class is a subclass of Constraint.
- Type definitions for any new procedure types used by class methods declared in the widget class part
- A structure part definition for the new fields that this widget class adds to its superclass's widget class structure.
- The complete widget class structure definition for this widget.
- The complete widget class extension structure definition for this widget, if any.
- The symbolic constant identifying the class extension version, if any.
- The name of the global class structure variable containing the generic class structure for this class.

- An inherit constant for each new procedure in the widget class part 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 representation types used by the Label widget */
#define XtRJustify "Justify"

/* New fields for the Label widget record */
typedef struct {
/* Settable resources */
        Pixel foreground;
        XFontStruct *font;
        String label;                   /* text to display */
        XtJustify justify;
        Dimension internal_width;       /* # pixels horizontal border */
        Dimension internal_height;      /* # 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;
        XtPointer 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 class is **ConstrainP.h**.

1.6.4. 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_inited*).
- 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 new operations specific to the widget).

The *superclass* field points to the superclass global class record, declared in the superclass private .h file. 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 1.6.7, 1.6.9, and 1.6.10).

The *class_name* field contains the text name for this class, which is 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. This string must be permanently allocated prior to or during the execution of the class initialization procedure and must not be subsequently deallocated.

The *widget_size* field is the size of the corresponding widget instance structure (not the size of the class structure).

The *version* field indicates the toolkit implementation version number and is used for runtime consistency checking of the X Toolkit and widgets in an application. Widget writers must set it to the implementation-defined symbolic value **XtVersion** in the widget class structure initialization. Those widget writers who believe that their widget binaries are compatible with other implementations of the Intrinsics can put the special value **XtVersionDontCheck** in the *version* field to disable version checking for those widgets. If a widget needs to compile alternative code for different revisions of the Intrinsics interface definition, it may use the symbol **XtSpecificationRelease**, as described in Chapter 13. Use of **XtVersion** allows the Intrinsics implementation to recognize widget binaries that were compiled with older implementations.

The *extension* field is for future upward compatibility. If the widget programmer adds 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 .c file also contains the declaration of the global class structure pointer variable used to create instances of the class. The following is an abbreviated version of the .c file for a Label widget. The resources table is described in Chapter 9.

```
/* 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},
                        .
                        .
                        .
}

/* 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)&coreClassRec,
        /* 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 LabelSetText(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 */
        .
        .
        .
```

1.6.5. Widget Class and Superclass Look Up

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

WidgetClass XtClass(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Object or any subclass thereof.

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. Must be of class Object or any subclass thereof.

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

1.6.6. Widget Subclass Verification

To check the subclass to which a widget belongs, use **XtIsSubclass**.

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

w Specifies the widget or object instance whose class is to be checked. Must be of class Object or any subclass thereof.

widget_class Specifies the widget class for which to test. Must be **objectClass** or any subclass thereof.

The **XtIsSubclass** function returns **True** if the class of the specified widget is equal to or is a subclass of the specified class. The widget's class can be any number of subclasses down the chain and need not be an immediate subclass of the specified 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 test if a given widget belongs to a subclass of an Intrinsics-defined class, the Intrinsics define macros or functions equivalent to **XtIsSubclass** for each of the built-in classes. These procedures are **XtIsObject**, **XtIsRectObj**, **XtIsWidget**, **XtIsComposite**, **XtIsConstraint**, **XtIsShell**, **XtIsOverrideShell**, **XtIsWMShell**, **XtIsVendorShell**, **XtIsTransientShell**, **XtIsTopLevelShell** and **XtIsApplicationShell**.

All these macros and functions have the same argument description.

Boolean XtIs<*class*> (*w*)
  Widget *w*;

w Specifies the widget or object instance whose class is to be checked. Must be of class Object or any subclass thereof.

These procedures may be faster than calling **XtIsSubclass** directly for the built-in classes.

To check a widget's class and to generate a debugging error message, use **XtCheckSubclass**, defined in <**X11/IntrinsicP.h**>:

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

w Specifies the widget or object whose class is to be checked. Must be of class Object or any subclass thereof.

widget_class Specifies the widget class for which to test. Must be **objectClass** or any subclass thereof.

message Specifies the message to be used.

The **XtCheckSubclass** macro determines if the class of the specified widget is equal to or is a subclass of the specified class. The widget's class can be any number of subclasses down the chain and need not be an immediate subclass of the specified class. If the specified widget's class 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 module has been compiled with the compiler symbol DEBUG defined; otherwise, it is defined as the empty string and generates no code.

1.6.7. Superclass Chaining

While most fields in a widget class structure are self-contained, some fields are linked to their corresponding fields in their superclass structures. With a linked field, the Intrinsics access the field's 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 are

In all widget classes:	*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*
In Composite widget classes:	*geometry_manager* *change_managed* *insert_child* *delete_child* *accepts_objects*
In Constraint widget classes:	*constraint_size*
In Shell widget classes:	*root_geometry_manager*

With downward superclass chaining, the invocation of an operation first accesses the field from the Object, RectObj, and Core class structures, then from 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 following fields of the **ConstraintClassPart** and **ConstraintClassExtensionRec** structures are chained from the Constraint class down to the subclass:

resources

initialize
set_values
get_values_hook

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

destroy
actions

For subclasses of Constraint, the following field of **ConstraintClassPart** is chained from the subclass up to the Constraint class:

destroy

1.6.8. Class Initialization: class_initialize and class_part_initialize Procedures

Many class records can be initialized completely at compile or link time. In some cases, however, a class may need to register type converters or perform other sorts of once-only runtime 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 Intrinsics. A class initialization procedure pointer is of type **XtProc**:

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

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

In addition to the class initialization that is done exactly once, some classes perform initialization for fields in their parts of the class record. These are performed not just for the particular class but for subclasses as well, and are done in the class's class part initialization procedure, a pointer to which is stored in the *class_part_initialize* field. The class_part_initialize procedure pointer is of type **XtWidgetClassProc**.

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

widget_class Points to the class structure for the class being initialized.

During class initialization, the class part initialization procedures 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 procedure. 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, Composite's class_part_initialize procedure is called to initialize the composite part of C's class record. Finally, the class_part_initialize procedures for A, B, and C, in that order, are called. For further information, see Section 1.6.9. 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 a nonzero value. After the one-time initialization, a class structure is constant.

The following example provides the class initialization procedure for a Label class.

```
static void ClassInitialize()
{
        XtSetTypeConverter(XtRString, XtRJustify, CvtStringToJustify,
                                    NULL, 0, XtCacheNone, NULL);
}
```

1.6.9. Initializing a Widget Class

A class is initialized when the first widget of that class or any subclass is created. To initialize a widget class without creating any widgets, use **XtInitializeWidgetClass**.

void XtInitializeWidgetClass(*object_class*)
    WidgetClass *object_class*;

object_class Specifies the object class to initialize. May be **objectClass** or any subclass thereof.

If the specified widget class is already initialized, **XtInitializeWidgetClass** returns immediately.

If the class initialization procedure registers type converters, these type converters are not available until the first object of the class or subclass is created or **XtInitializeWidgetClass** is called (see Section 9.6).

1.6.10. 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 **XtInherit***Xyz* 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 chained operations declared in Core and Constraint records are never inherited. 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, called the inheritance constant, is usually the Intrinsics internal value **_XtInherit** cast to the appropriate type. **_XtInherit** is a procedure that issues an error message if it is actually called.

For example, **CompositeP.h** contains these definitions:

```
#define XtInheritGeometryManager ((XtGeometryHandler) _XtInherit)
#define XtInheritChangeManaged ((XtWidgetProc) _XtInherit)
```

```
#define XtInheritInsertChild ((XtArgsProc) _XtInherit)
#define XtInheritDeleteChild ((XtWidgetProc) _XtInherit)
```

Composite's class_part_initialize procedure begins as follows:

```
static void CompositeClassPartInitialize(widgetClass)
    WidgetClass widgetClass;
{
    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;
    }
    .
    .
    .
```

Nonprocedure fields may be inherited in the same manner as procedure fields. The class may declare any reserved value it wishes for the inheritance constant for its new fields. The following inheritance constants are defined:

For Core:

XtInheritRealize

XtInheritResize

XtInheritExpose

XtInheritSetValuesAlmost

XtInheritAcceptFocus

XtInheritQueryGeometry

XtInheritTranslations

XtInheritDisplayAccelerator

For Composite:

XtInheritGeometryManager

XtInheritChangeManaged

XtInheritInsertChild

XtInheritDeleteChild

For Shell:

XtInheritRootGeometryManager

1.6.11. Invocation of Superclass Operations

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

Note

> A class method should not use **XtSuperclass** but should instead call the class method of its own specific superclass directly through the superclass record. That is, it should use its own class pointers only, not the widget's class pointers, as the widget's class may be a subclass of the class whose implementation is being referenced.

This technique is referred to as *enveloping* the superclass's operation.

1.6.12. Class Extension Records

It may be necessary at times to add new fields to already existing widget class structures. To permit this to be done without requiring recompilation of all subclasses, the last field in a class part structure should be an extension pointer. If no extension fields for a class have yet been defined, subclasses should initialize the value of the extension pointer to NULL.

If extension fields exist, as is the case with the Composite, Constraint and Shell classes, subclasses can provide values for these fields by setting the *extension* pointer for the appropriate part in their class structure to point to a statically declared extension record containing the additional fields. Setting the *extension* field is never mandatory; code that uses fields in the extension record must always check the *extension* field and take some appropriate default action if it is NULL.

In order to permit multiple subclasses and libraries to chain extension records from a single *extension* field, extension records should be declared as a linked list and each extension record definition should contain the following four fields at the beginning of the structure declaration:

```
struct {
        XtPointer next_extension;
        XrmQuark record_type;
        long version;
        Cardinal record_size;
};
```

next_extension	Specifies the next record in the list, or NULL.
record_type	Specifies the particular structure declaration to which each extension record instance conforms.
version	Specifies a version id symbolic constant supplied by the definer of the structure.
record_size	Specifies the total number of bytes allocated for the extension record.

The *record_type* field identifies the contents of the extension record and is used by the definer of the record to locate its particular extension record in the list. The *record_type* field is normally assigned the result of **XrmStringToQuark** for a registered string constant. The Intrinsics reserve all record type strings beginning with the two characters ‘‘XT’’ for future standard uses. The value **NULLQUARK** may also be used by the class part owner in extension records attached to its own class part extension field to identify the extension record unique to that particular class.

The *version* field is an owner-defined constant that may be used to identify binary files that have been compiled with alternate definitions of the remainder of the extension record data structure. The private header file for a widget class should provide a symbolic constant for subclasses to use to initialize this field. The *record_size* field value includes the four common header fields and should normally be initialized with **sizeof**().

Any value stored in the class part extension fields of **CompositeClassPart**, **ConstraintClassPart**, or **ShellClassPart** must point to an extension record conforming to this definition.

Chapter 2

Widget Instantiation

A hierarchy 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 the 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, that is, members of the class **compositeWidgetClass**, are containers for an arbitrary but widget 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 classes 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 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 upward traversal of the widget tree, each widget has a pointer to its parent widget. The Shell widget that **XtAppCreateShell** returns has a *parent* pointer of NULL.

To facilitate downward traversal of the widget tree, the *children* field of each composite widget is a pointer to an array of child 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.

2.1. Initializing the X Toolkit

Before an application can call any Intrinsics function other than **XtSetLanguageProc**, it must initialize the Intrinsics by using

- **XtToolkitInitialize**, which initializes the Intrinsics internals.
- **XtCreateApplicationContext**, which initializes the per-application state.
- **XtDisplayInitialize** or **XtOpenDisplay**, which initializes the per-display state.
- **XtAppCreateShell**, which creates the root of a widget tree.

or an application can call the convenience procedure **XtAppInitialize** which combines the functions of the preceding procedures. An application wishing to use the ANSI C locale mechanism should call **XtSetLanguageProc** prior to calling **XtDisplayInitialize**, **XtOpenDisplay**, or **XtAppInitialize**.

Multiple instances of X Toolkit applications may be implemented 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 instance may need multiple display connections to have widgets on multiple displays. From the application's point of view, multiple display connections usually are treated together as a single unit for purposes of event dispatching. To accommodate both requirements, the Intrinsics 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 one or more X **Display** pointers for that application. The Intrinsics handle all display connections within a single application context simultaneously, handling input in a round-robin fashion. The application context type **XtAppContext** is opaque to clients.

To initialize the Intrinsics 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 remaining display connections 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, in a callback procedure), **XtDestroyApplicationContext** does not destroy the application context until the dispatch is complete.

To get the application context in which a given widget was created, use **XtWidgetToApplicationContext**.

XtAppContext XtWidgetToApplicationContext(*w*)
    Widget *w*;

w Specifies the widget for which you want the application context. Must be of class Object or any subclass thereof.

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*;
int **argc*;
String **argv*;

app_context	Specifies the application context.
display	Specifies a previously opened display connection. Note that a single display connection 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 Section 15.9 in *Xlib – C Language X Interface* and Section 2.4 of this specification.
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 list of command line parameters.

The **XtDisplayInitialize** function retrieves the language string to be used for the specified display (see Section 11.11), calls the language procedure (if set) with that language string, builds the resource database for the default screen, 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 POSIX-based systems, the application name is usually the final component of *argv*[0]. If the synchronous resource is **True**, **XtDisplayInitialize** calls the Xlib **XSynchronize** function to put Xlib into synchronous mode for this display connection and any others currently open in the application context. See Sections 2.3 and 2.4 for details on the *application_name*, *application_class*, *options*, and *num_options* arguments.

XtDisplayInitialize calls **XrmSetDatabase** to associate the resource database of the default screen with the display before returning.

To open a display, initialize it, and then 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*;
String *application_class*;
XrmOptionDescRec **options*;
Cardinal *num_options*;
int **argc*;
String **argv*;

app_context	Specifies the application context.
display_string	Specifies the display string, or NULL.
application_name	Specifies the name of the application instance, or NULL.
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**.
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 list of command line parameters.

The **XtOpenDisplay** function calls **XOpenDisplay** with the specified *display_string*. If *display_string* is NULL, **XtOpenDisplay** uses the current value of the –display option specified in *argv*. If no display is specified in *argv*, the user's default display is retrieved from the environment. On POSIX-based systems, this is the value of the **DISPLAY** environment variable.

If this succeeds, **XtOpenDisplay** then calls **XtDisplayInitialize** and passes it the opened display and the value of the –name option specified in *argv* as the application name. If no –name option is specified and *application_name* is non-NULL, *application_name* is passed to **XtDisplayInitialize**. If *application_name* is NULL and if the environment variable **RESOURCE_NAME** is set, the value of **RESOURCE_NAME** is used. Otherwise, the application name is the name used to invoke the program. On implementations that conform to ANSI C Hosted Environment support, the application name will be *argv*[0] less any directory and file type components, that is, the final component of *argv*[0], if specified. If *argv*[0] does not exist or is the empty string, the application name is ''main''. **XtOpenDisplay** returns the newly opened display or NULL if it failed.

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 calls **XCloseDisplay** with 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. Establishing the Locale

Resource databases are specified to be created in the current process locale. During display initialization prior to creating the per-screen resource database, the Intrinsics will call out to a specified application procedure to set the locale according to options found on the command line or in the per-display resource specifications.

The callout procedure provided by the application is of type **XtLanguageProc**.

typedef String (*XtLanguageProc)(Display*, String, XtPointer);
    Display **display*;
    String *language*;
    XtPointer *client_data*;

display Passes the display.

language Passes the initial language value obtained from the command line or server per-display resource specifications.

client_data Passes the additional client data specified in the call to **XtSetLanguageProc**.

The language procedure allows an application to set the locale to the value of the language resource determined by **XtDisplayInitialize**. The function returns a new language string that will be subsequently used by **XtDisplayInitialize** to establish the path for loading resource files. The returned string will be copied by the Intrinsics into new memory.

Initially, no language procedure is set by the Intrinsics. To set the language procedure for use by **XtDisplayInitialize** use **XtSetLanguageProc**.

XtLanguageProc XtSetLanguageProc(*app_context*, *proc*, *client_data*)
    XtAppContext *app_context*;
    XtLanguageProc *proc*;
    XtPointer *client_data*;

app_context Specifies the application context in which the language procedure is to be used, or NULL.

proc Specifies the language procedure.

client_data Specified additional client data to be passed to the language procedure when it is called.

XtSetLanguageProc sets the language procedure that will be called from **XtDisplayInitialize** for all subsequent Displays initialized in the specified application context. If *app_context* is NULL, the specified language procedure is registered in all application contexts created by the calling process, including any future application contexts that may be created. If *proc* is NULL a default language procedure is registered. **XtSetLanguageProc** returns the previously registered language procedure. If a language procedure has not yet been registered, the return value is unspecified but if this return value is used in a subsequent call to **XtSetLanguageProc**, it will cause the default language procedure to be registered.

The default language procedure does the following:

- Sets the locale according to the environment. On ANSI C-based systems this is done by calling **setlocale**(**LC_ALL**, *language*). If an error is encountered a warning message is issued with **XtWarning**.
- Calls **XSupportsLocale** to verify that the current locale is supported. If the locale is not supported, a warning message is issued with **XtWarning** and the locale is set to ‘‘C’’.
- Calls **XSetLocaleModifiers** specifying the empty string.

- Returns the value of the current locale. On ANSI C-based systems this is the return value from a final call to **setlocale**(**LC_ALL**, NULL).

A client wishing to use this mechanism to establish locale can do so by calling **XtSetLanguageProc** prior to **XtDisplayInitialize**, as in the following example.

```
Widget top;
XtSetLanguageProc(NULL, NULL, NULL);
top = XtAppInitialize( ... );
...
```

2.3. Loading the Resource Database

The **XtDisplayInitialize** function first determines the language string to be used for the specified display. It then creates a resource database for the default screen of the display by combining the following sources in order, with the entries in the first named source having highest precedence:

- Application command line (*argc*, *argv*).
- Per-host user environment resource file on the local host.
- Per-screen resource specifications from the server.
- Per-display resource specifications from the server or from the user preference file on the local host.
- Application-specific user resource file on the local host.
- Application-specific class resource file on the local host.

When the resource database for a particular screen on the display is needed (either internally, or when **XtScreenDatabase** is called), it is created in the following manner using the sources listed above in the same order:

- A temporary database, the ‘‘server resource database’’, is created from the string returned by **XResourceManagerString** or, if **XResourceManagerString** returns NULL, the contents of a resource file in the user’s home directory. On POSIX-based systems, the usual name for this user preference resource file is $HOME/**.Xdefaults**.

- If a language procedure has been set, **XtDisplayInitialize** first searches the command line for the option ‘‘-xnlLanguage’’, or for a -xrm option that specifies the xnlLanguage/XnlLanguage resource, as specified by Section 2.4. If such a resource is found, the value is assumed to be entirely in XPCS, the X Portable Character Set. If neither option is specified on the command line, **XtDisplayInitialize** queries the server resource database (which is assumed to be entirely in XPCS) for the resource *name***.xnlLanguage**, class *Class***.XnlLanguage** where *name* and *Class* are the *application_name* and *application_class* specified to **XtDisplayInitialize**. The language procedure is then invoked with the resource value if found, else the empty string. The string returned from the language procedure is saved for all future references in the Intrinsics that require the per-display language string.

- The screen resource database is initialized by parsing the command line in the manner specified by Section 2.4.

- If a language procedure has not been set, the initial database is then queried for the resource *name***.xnlLanguage**, class *Class***.XnlLanguage** as specified above. If this database query fails, the server resource database is queried; if this query also fails, the language is determined from the environment; on POSIX-based systems, this is done by retrieving the value of the **LANG** environment variable. If no language string is found, the empty string is used. This language string is saved for all future references in the Intrinsics that require the per-display language string.

- After determining the language string, the user's environment resource file is then merged into the initial resource database if the file exists. This file is user-, host-, and process-specific and is expected to contain user preferences that are to override those specifications in the per-display and per-screen resources. On POSIX-based systems, the user's environment resource file name is specified by the value of the **XENVIRONMENT** environment variable. If this environment variable does not exist, the user's home directory is searched for a file named **.Xdefaults-***host*, where *host* is the host name of the machine on which the application is running.

- The per-screen resource specifications are then merged into the screen resource database, if they exist. These specifications are the string returned by **XScreenResourceString** for the respective screen and are owned entirely by the user.

- Next, the server resource database created earlier is merged into the screen resource database. The server property, and corresponding user preference file, are owned and constructed entirely by the user.

- The application-specific user resource file from the local host is then merged into the screen resource database. This file contains user customizations and is stored in a directory owned by the user. Either the user or the application or both can store resource specifications in the file. Each should be prepared to find and respect entries made by the other. The file name is found by calling **XrmSetDatabase** with the current screen resource database, after preserving the original display-associated database, then calling **XtResolvePathname** with the parameters (*display*, NULL, NULL, NULL, *path*, NULL, 0, NULL) where *path* is defined in an operating-system-specific way. On POSIX-based systems, *path* is defined to be the value of the environment variable **XUSERFILESEARCHPATH** if this is defined. If **XUSERFILESEARCHPATH** is not defined, an implementation-dependent default value is used. This default value is constrained in the following manner:

  – If the environment variable **XAPPLRESDIR** is not defined, the default **XUSERFILESEARCHPATH** must contain at least six entries. These entries must contain $HOME as the directory prefix, plus the following substitutions:

    1. %C, %N, %L or %C, %N, %l, %t, %c
    2. %C, %N, %l
    3. %C, %N
    4. %N, %L or %N, %l, %t, %c
    5. %N, %l
    6. %N

    The order of these six entries within the path must be as given above. The order and use of substitutions within a given entry is implementation dependent.

  – If **XAPPLRESDIR** is defined, the default **XUSERFILESEARCHPATH** must contain at least seven entries. These entries must contain the following directory prefixes and substitutions:

1.	$XAPPLRESDIR	with	%C, %N, %L	or	%C, %N, %l, %t, %c
2.	$XAPPLRESDIR	with	%C, %N, %l		
3.	$XAPPLRESDIR	with	%C, %N		
4.	$XAPPLRESDIR	with	%N, %L	or	%N, %l, %t, %c
5.	$XAPPLRESDIR	with	%N, %l		
6.	$XAPPLRESDIR	with	%N		
7.	$HOME	with	%N		

The order of these seven entries within the path must be as given above. The order and use of substitutions within a given entry is implementation dependent.

- Lastly, the application-specific class resource file from the local host is merged into the screen resource database. This file is owned by the application and is usually installed in a system directory when the application is installed. It may contain site-wide customizations specified by the system manager. The name of the application class resource file is found by calling **XtResolvePathname** with the parameters (*display*, ‘‘app-defaults’’, NULL, NULL, NULL, NULL, 0, NULL). This file is expected to be provided by the developer of the application and may be required for the application to function properly. A simple application that wants to be assured of having a minimal set of resources in the absence of its class resource file can declare fallback resource specifications with **XtAppSetFallbackResources**. Note that the customization substitution string is retrieved dynamically by **XtResolvePathname** so that the resolved file name of the application class resource file can be affected by any of the earlier sources for the screen resource database, even though the contents of the class resource file have lowest precedence. After calling **XtResolvePathname**, the original display-associated database is restored.

To obtain the resource database for a particular screen, use **XtScreenDatabase**.

```
XrmDatabase XtScreenDatabase(screen)
      Screen *screen;
```

screen Specifies the screen whose resource database is to be returned.

The **XtScreenDatabase** function returns the fully merged resource database as specified above, associated with the specified screen. If the specified *screen* does not belong to a **Display** initialized by **XtDisplayInitialize**, the results are undefined.

To obtain the default resource database associated with a particular display, use **XtDatabase**.

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

display Specifies the display.

The **XtDatabase** function is equivalent to **XrmGetDatabase**. It returns the database associated with the specified display, or NULL if a database has not been set.

To specify a default set of resource values that will be used to initialize the resource database if no application-specific class resource file is found (the last of the six sources listed above), use **XtAppSetFallbackResources**.

```
void XtAppSetFallbackResources(app_context, specification_list)
      XtAppContext app_context;
      String *specification_list;
```

app_context Specifies the application context in which the fallback specifications will be used.

specification_list Specifies a NULL-terminated list of resource specifications to preload the database, or NULL.

Each entry in *specification_list* points to a string in the format of **XrmPutLineResource**. Following a call to **XtAppSetFallbackResources**, when a resource database is being created for a particular screen and the Intrinsics are not able to find or read an application-specific class resource file according to the rules given above and if *specification_list* is not NULL the resource specifications in *specification_list* will be merged into the screen resource database in place of the application-specific class resource file. **XtAppSetFallbackResources** is not required to copy *specification_list*; the caller must ensure that the contents of the list and of the strings addressed by the list remain valid until all displays are initialized or until **XtAppSetFallbackResources** is called again. The value NULL for *specification_list* removes any previous fallback resource specification for the application context. The intended use for fallback resources is to provide a minimal number of resources that will make the application usable (or at least terminate with helpful diagnostic messages) when some problem exists in finding and loading the application defaults file.

2.4. Parsing the Command Line

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

–display Specifies the display name for **XOpenDisplay**.

–name Sets the resource name prefix, which overrides the application name passed to **XtOpenDisplay**.

–xnllanguage Specifies the initial language string for establishing locale and for finding application class resource files.

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 described in Section 15.9 in *Xlib – C Language X Interface*.

```
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 */
        XrmoptionResArg,         /* Use the next argument as input to XrmPutLineResource*/
        XrmoptionSkipArg,        /* Ignore this option and the next argument in argv */
        XrmoptionSkipNArgs,      /* Ignore this option and the next */
                                 /* OptionDescRec.value arguments 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;   /* Location of the resource value */
        XPointer 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	.synchronous	NoArg	‘‘on’’
+synchronous	.synchronous	NoArg	‘‘off’’
–title	.title	SepArg	next argument
–xnllanguage	.xnlLanguage	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 **True**, the values of **XtDefaultForeground** and **XtDefaultBackground** are exchanged for all screens on the Display.

The value of the synchronous resource specifies whether or not Xlib is put into synchronous mode. If a value is found in the resource database during display initialization, **XtDisplayInitialize** makes a call to **XSynchronize** for all display connections currently open in the application context. Therefore, when multiple displays are initialized in the same application context, the most recent value specified for the synchronous resource is used for all displays in the application context.

The value of the selectionTimeout resource applies to all displays opened in the same application context. When multiple displays are initialized in the same application context, the most recent value specified is used for all displays in the application context.

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 resource 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. The Intrinsics reserve all option names beginning with the characters ‘‘-xt’’ for future standard uses.

2.5. 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, which then are 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 parents' 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** with the top-level widget to execute the second and third phases. **XtRealizeWidget** first recursively traverses the widget tree in a postorder (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 preorder (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.

2.5.1. Creating and Merging Argument Lists

Many Intrinsics functions may be passed pairs of resource names and values. These are passed as an arglist, a pointer to an array of **Arg** structures, which contains

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

where **XtArgVal** is as defined in Section 1.5.

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 in *value*.

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

void 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**, else 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 expressions with side effects such as auto-increment or auto-decrement within the first argument to **XtSetArg**. **XtSetArg** can be implemented as a macro that evaluates the first argument twice.

To merge two arglist arrays, use **XtMergeArgLists**.

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

args1 Specifies the first argument list.

num_args1 Specifies the number of entries in the first argument list.

args2 Specifies the second argument list.

num_args2 Specifies the number of entries in the second argument list.

The **XtMergeArgLists** function allocates enough storage to hold the combined arglist arrays and copies them into it. Note that it does not check for duplicate entries. The length of the returned list is the sum of the lengths of the specified lists. When it is no longer needed, free the returned storage by using **XtFree**.

All Intrinsics interfaces that require **ArgList** arguments have analogs conforming to the ANSI C variable argument list (traditionally called ‘‘varargs’’) calling convention. The name of the analog is formed by prefixing ‘‘Va’’ to the name of the corresponding **ArgList** procedure; e.g., -**XtVaCreateWidget**. Each procedure named **XtVa***something* takes as its last arguments, in place of the corresponding **ArgList**/ **Cardinal** parameters, a variable parameter list of resource name and value pairs where each name is of type **String** and each value is of type **XtArgVal**. The end of the list is identified by a *name* entry containing NULL. Developers writing in the C language wishing to pass resource name and value pairs to any of these interfaces may use the **ArgList** and varargs forms interchangeably.

Two special names are defined for use only in varargs lists: **XtVaTypedArg** and **XtVaNestedList**.

```
#define XtVaTypedArg "XtVaTypedArg"
```

If the name **XtVaTypedArg** is specified in place of a resource name, then the following four arguments are interpreted as a *name/type/value/size* tuple *where* name is of type **String**, *type* is of type **String**, *value* is of type **XtArgVal**, and *size* is of type int. When a varargs list containing **XtVaTypedArg** is processed, a resource type conversion (see Section 9.6) is performed if necessary to convert the value into the format required by the associated resource. If *type* is XtRString then *value* contains a pointer to the string and *size* contains the number of bytes allocated, including the trailing null byte. If *type* is not XtRString, then *if* size is less than or equal to **sizeof**(**XtArgVal**), the value should be the data cast to the type **XtArgVal**, otherwise *value* is a pointer to the data. If the type conversion fails for any reason, a warning message is issued and the list entry is skipped.

```
#define XtVaNestedList  "XtVaNestedList"
```

If the name **XtVaNestedList** is specified in place of a resource name, then the following argument is interpreted as an **XtVarArgsList** value, which specifies another varargs list that is logically inserted into the original list at the point of declaration. The end of the nested list is identified with a name entry containing NULL. Varargs lists may nest to any depth.

To dynamically allocate a varargs list for use with **XtVaNestedList** in multiple calls, use **XtVaCreateArgsList**.

```
typedef XtPointer XtVarArgsList;
```

XtVarArgsList XtVaCreateArgsList(*unused*, ...)
    XtPointer *unused*;

unused This argument is not currently used and must be specified as NULL.

... Specifies a variable parameter list of resource name and value pairs.

The **XtVaCreateArgsList** function allocates memory and copies its arguments into a single list pointer, which may be used with **XtVaNestedList**. The end of both lists is identified by a *name* entry containing NULL. Any entries of type **XtVaTypedArg** are copied as specified without applying conversions. Data passed by reference (including Strings) are not copied, only the pointers themselves; the caller must ensure that the data remain valid for the lifetime of the created varargs list. The list should be freed using **XtFree** when no longer needed.

Use of resource files and the resource database is generally encouraged over lengthy arglist or varargs lists whenever possible in order to permit modification without recompilation.

2.5.2. Creating a Widget Instance

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

Widget XtCreateWidget(*name*, *object_class*, *parent*, *args*, *num_args*)
    String *name*;
    WidgetClass *object_class*;
    Widget *parent*;
    ArgList *args*;
    Cardinal *num_args*;

name Specifies the resource instance 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 the same parent.

object_class Specifies the widget class pointer for the created object. Must be **objectClass** or any subclass thereof.

parent Specifies the parent widget. Must be of class Object or any subclass thereof.

args Specifies the argument list to override any other resource specifications.

num_args Specifies the number of entries in the argument list.

The **XtCreateWidget** function performs all the boilerplate operations of widget creation, doing the following in order:

- 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.
- If the specified class is not **coreWidgetClass** or a subclass thereof, and the parent's class is a subclass of **compositeWidgetClass** and either no extension record in the parent's composite class part extension field exists with the *record_type* **NULLQUARK** or the *accepts_objects* field in the extension record is **False**, **XtCreateWidget** issues a fatal error; see Section 3.1 and Chapter 12.
- Allocates memory for the widget instance.
- If the parent is a member of the class **constraintWidgetClass**, 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 **CoreClassPart** resource lists specified for this class and all superclasses.
- If the parent is a member of the class **constraintWidgetClass**, initializes the resource fields of the constraints record by using the **ConstraintClassPart** resource lists specified for the parent's class and all superclasses up to **constraintWidgetClass**.
- Calls the initialize procedures for the widget starting at the Object initialize procedure on down to the widget's initialize procedure.
- If the parent is a member of the class **compositeWidgetClass**, puts the widget into its parent's children list by calling its parent's insert_child procedure. For further information, see Section 3.1.
- If the parent is a member of the class **constraintWidgetClass**, calls the **ConstraintClassPart** initialize procedures, starting at **constraintWidgetClass** on down to the parent's **ConstraintClassPart** initialize procedure.

To create an instance of a widget using varargs lists, use **XtVaCreateWidget**.

Widget XtVaCreateWidget(*name*, *object_class*, *parent*, ...)
    String *name*;
    WidgetClass *object_class*;
    Widget *parent*;

name Specifies the resource name for the created widget.

object_class Specifies the widget class pointer for the created object. Must be **objectClass** or any subclass thereof.

parent Specifies the parent widget. Must be of class Object or any subclass thereof.

... Specifies the variable argument list to override any other resource specifications.

The **XtVaCreateWidget** procedure is identical in function to **XtCreateWidget** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

2.5.3. Creating an Application Shell Instance

An application can have multiple top-level widgets, each of which specifies a unique widget tree which can potentially be on different screens or displays. An application uses **XtAppCreateShell** to create independent widget trees.

Widget XtAppCreateShell(*name*, *application_class*, *widget_class*, *display*,
args, *num_args*)
String *name*;
String *application_class*;
WidgetClass *widget_class*;
Display **display*;
ArgList *args*;
Cardinal *num_args*;

name Specifies the instance name of the shell widget. If *name* is NULL, the application name passed to **XtDisplayInitialize** is used.

application_class Specifies the resource class string to be used in place of the widget *class_name* string when *widget_class* is **applicationShellWidgetClass** or a subclass thereof.

widget_class Specifies the widget class for the top-level widget (e.g., **applicationShellWidgetClass**)

display Specifies the display for the default screen and for the resource database used to retrieve the shell widget resources.

args Specifies the argument list to override any other resource specifications.

num_args Specifies the number of entries in the argument list.

The **XtAppCreateShell** function creates a new shell widget instance as the root of a widget tree. The screen resource for this widget is determined by first scanning *args* for the XtNscreen argument. If no XtNscreen argument is found, the resource database associated with the default screen of the specified display is queried for the resource *name*.screen, class *Class*.Screen where *Class* is the specified *application_class* if *widget_class* is **applicationShellWidgetClass** or a subclass thereof. If *widget_class* is not **applicationShellWidgetClass** or a subclass, *Class* is the *class_name* field from the **CoreClassPart** of the specified *widget_class*. If this query fails, the default screen of the specified display is used. Once the screen is determined, the resource database associated with that screen is used to retrieve all remaining resources for the shell widget not specified in *args*. The widget name and *Class* as determined above are used as the leftmost (i.e., root) components in all fully qualified resource names for objects within this widget tree.

If the specified widget class is a subclass of WMShell, the name and *Class* as determined above will be stored into the **WM_CLASS** property on the widget's window when it becomes realized. If the specified *widget_class* is **applicationShellWidgetClass** or a subclass thereof the **WM_COMMAND** property will also be set from the values of the XtNargv and XtNargc resources.

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)

To create a top-level widget that is the root of a widget tree using varargs lists, use **XtVaAppCreateShell**.

Widget XtVaAppCreateShell(*name*, *application_class*, *widget_class*, *display*, ...)
  String *name*;
  String *application_class*;
  WidgetClass *widget_class*;
  Display **display*;

name Specifies the instance name of the shell widget. If *name* is NULL, the application name passed to **XtDisplayInitialize** is used.

application_class
Specifies the resource class string to be used in place of the widget *class_name* string when *widget_class* is **applicationShellWidgetClass** or a subclass thereof.

widget_class Specifies the widget class for the top-level widget.

display Specifies the display for the default screen and for the resource database used to retrieve the shell widget resources.

... Specifies the variable argument list to override any other resource specifications.

The **XtVaAppCreateShell** procedure is identical in function to **XtAppCreateShell** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

2.5.4. Convenience Procedure to Initialize an Application

To initialize the Intrinsics internals, create an application context, open and initialize a display, and create the initial application shell instance, an application may use **XtAppInitialize** or **XtVaAppInitialize**.

Widget XtAppInitialize(*app_context_return*, *application_class*, *options*, *num_options*,
      argc_in_out, *argv_in_out*, *fallback_resources*, *args*, *num_args*)
  XtAppContext **app_context_return*;
  String *application_class*;
  XrmOptionDescList *options*;
  Cardinal *num_options*;
  int **argc_in_out*;
  String **argv_in_out*;
  String **fallback_resources*;
  ArgList *args*;
  Cardinal *num_args*;

app_context_return Returns the application context, if non-NULL.

application_class Specifies the class name of the application.

options Specifies the command line options table.

num_options	Specifies the number of entries in *options*.
argc_in_out	Specifies a pointer to the number of command line arguments.
argv_in_out	Specifies a pointer to the command line arguments.
fallback_resources	Specifies resource values to be used if the application class resource file cannot be opened or read, or NULL.
args	Specifies the argument list to override any other resource specifications for the created shell widget.
num_args	Specifies the number of entries in the argument list.

The **XtAppInitialize** function calls **XtToolkitInitialize** followed by **XtCreateApplicationContext**, then calls **XtOpenDisplay** with *display_string* NULL and *application_name* NULL, and finally calls **XtAppCreateShell** with *application_name* NULL, *widget_class* **applicationShellWidgetClass**, and the specified *args* and *num_args* and returns the created shell. The modified *argc* and *argv* returned by **XtDisplayInitialize** are returned in *argc_in_out* and *argv_in_out*. If *app_context_return* is not NULL, the created application context is also returned. If the display specified by the command line cannot be opened, an error message is issued and **XtAppInitialize** terminates the application. If *fallback_resources* is non-NULL, **XtAppSetFallbackResources** is called with the value prior to calling **XtOpenDisplay**.

```
Widget XtVaAppInitialize(app_context_return, application_class, options, num_options,
                argc_in_out, argv_in_out, fallback_resources, ...)
   XtAppContext *app_context_return;
   String application_class;
   XrmOptionDescList options;
   Cardinal num_options;
   int *argc_in_out;
   String *argv_in_out;
   String *fallback_resources;
```

app_context_return	Returns the application context, if non-NULL.
application_class	Specifies the class name of the application.
options	Specifies the command line options table.
num_options	Specifies the number of entries in *options*.
argc_in_out	Specifies a pointer to the number of command line arguments.
argv_in_out	Specifies the command line arguments array.
fallback_resources	Specifies resource values to be used if the application class resource file cannot be opened, or NULL.
...	Specifies the variable argument list to override any other resource specifications for the created shell.

The **XtVaAppInitialize** procedure is identical in function to **XtAppInitialize** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

2.5.5. Widget Instance Initialization: the initialize Procedure

The initialize procedure pointer in a widget class is of type **XtInitProc**.

```
typedef void (*XtInitProc)(Widget, Widget, ArgList, Cardinal*);
     Widget request;
     Widget new;
     ArgList args;
     Cardinal *num_args;
```

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

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

args Specifies the argument list passed by the client, for computing derived resource values. If the client created the widget using a varargs form, any resources specified via **XtVaTypedArg** are converted to the widget representation and the list is transformed into the **ArgList** format.

num_args Specifies the number of entries in the argument list.

An initialization procedure performs the following:

- Allocates space for and copies any resources referenced by address that the client is allowed to free or modify after the widget has been created. For example, if a widget has a field that is a **String**, it may choose not to depend on the characters at that address remaining constant but dynamically allocate space for the string and copy it to the new space. Widgets that do not copy one or more resources referenced by address should clearly so state in their user documentation.

Note

> It is not necessary to allocate space for or to 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 its other resources.

Note

> A widget may only directly assign its own *width* and *height* within the initialize, initialize_hook, set_values and set_values_hook procedures; see Chapter 6.

- 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 colormap for a depth that does not support that colormap.

Initialization procedures are called in superclass-to-subclass order after all fields specified in the resource lists have been initialized. The initialize procedure does not need to examine *args* and *num_args* if all public resources are declared in the resource list. 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 a subclass to determine the difference between an explicitly specified field and a field computed by a superclass. The *request* widget is a copy of the widget as initialized by the arglist and resource database. 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. If the initialize procedure needs to call any routines that operate on a widget, it should specify *new* as the widget instance.

2.5.6. Constraint Instance Initialization: the ConstraintClassPart initialize Procedure

The constraint initialization procedure pointer, found in the **ConstraintClassPart** *initialize* field of the widget class record, is of type **XtInitProc**. The values passed to the parent constraint initialization procedures are the same as those passed to the child's class widget initialization procedures.

The *constraints* field of the *request* widget points to a copy of the constraints record as initialized by the arglist and resource database.

The constraint initialization procedure should compute any constraint fields derived from constraint resources. It can make further changes to the *new* widget to make the widget and any other constraint fields 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.

2.5.7. Nonwidget Data Initialization: the initialize_hook Procedure

Note

> The initialize_hook procedure is obsolete, as the same information is now available to the initialize procedure. The procedure has been retained for those widgets that used it in previous releases.

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 passed by the client. If the client created the widget using a varargs form, any resources specified via **XtVaTypedArg** are converted to the widget representation and the list is transformed into the **ArgList** format.

num_args Specifies the number of entries 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* field is NULL.

The initialize_hook procedure allows a widget instance to initialize nonresource data using information from the specified argument list as if it were a resource.

2.6. Realizing Widgets

To realize a widget instance, use **XtRealizeWidget**.

void XtRealizeWidget(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Core or any subclass thereof.

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 postorder traversal of the widget tree rooted at the specified widget and calls each non-NULL change_managed procedure of all composite widgets that have 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.

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, **XtVaCreateWidget**, **XtRealizeWidget**, **XtManageChildren**, **XtUnmanageChildren**, **XtUnrealizeWidget**, **XtSetMappedWhenManaged**, and **XtDestroyWidget** maintain the following invariants:

- If a composite widget is realized, then all its managed children are realized.
- If a composite widget is realized, then all its managed children that have *mapped_when_managed* **True** are mapped.

All Intrinsics functions and all widget routines should accept either realized or unrealized widgets. When calling the realize or change_managed procedures for children of a composite widget, **XtRealizeWidget** calls the procedures in reverse order of appearance in the **CompositePart** *children* list. By default, this ordering of the realize procedures will result in the stacking order of any newly created subwindows being top-to-bottom in the order of appearance on the list, and the most recently created child will be at the bottom.

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

Boolean XtIsRealized(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Object or any subclass thereof.

The **XtIsRealized** function returns **True** if the widget has been realized, that is, if the widget has a nonzero window ID. If the specified object is not a widget, the state of the nearest widget ancestor is returned.

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

2.6.1. 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 are used.

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

The realize procedure must create the widget's window.

Before calling the class realize procedure, the generic **XtRealizeWidget** function fills in a mask and a corresponding **XSetWindowAttributes** structure. It sets the following fields in *attributes* and corresponding bits in *value_mask* based on information in the widget core structure:

- The *background_pixmap* (or *background_pixel* if *background_pixmap* is **XtUnspecifiedPixmap**) is filled in from the corresponding field.
- The *border_pixmap* (or *border_pixel* if *border_pixmap* is **XtUnspecifiedPixmap**) is filled in from the corresponding field.
- The *colormap* is filled in from the corresponding field.
- The *event_mask* is filled in based on the event handlers registered, the event translations specified, whether the *expose* field is non-NULL, and whether *visible_interest* is **True**.
- The *bit_gravity* is set to **NorthWestGravity** if the *expose* field is NULL.

These or any 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 **coreWidgetClass** calls **XtCreateWindow** with the passed *value_mask* and *attributes* and with *window_class* and *visual* set to **CopyFromParent**. Both **compositeWidgetClass** and **constraintWidgetClass** inherit this realize procedure, and most new widget subclasses can do the same (see Section 1.6.10).

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 might set *bit_gravity* to **WestGravity**, **CenterGravity**, or **EastGravity**. Consequently, shrinking it would just move the bits appropriately, and no exposure event is needed for repainting.

If a composite widget's children should be realized in an order other than that specified (to control the stacking order, for example), it should call **XtRealizeWidget** on its children itself in the appropriate order from within its own realize procedure.

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

2.6.2. Window Creation Convenience Routine

Rather than call the Xlib **XCreateWindow** function explicitly, a realize procedure should normally 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 defines the additional window attributed. Must be of class Core or any subclass thereof.

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

visual Specifies the visual type (usually **CopyFromParent**).

value_mask Specifies which fields in the *attributes* structure are used.

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 widget core structure: *depth*, *screen*, *parent->core.window*, *x*, *y*, *width*, *height*, and *border_width*.

2.7. Obtaining Window Information from a Widget

The Core widget class definition contains the screen and window ids. The *window* field may be NULL for a while (see Sections 2.5 and 2.6).

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. Must be of class Core or any subclass thereof.

XtDisplay returns the display pointer for the specified widget.

Widget XtParent(*w*)
  Widget *w*;

w Specifies the widget. Must be of class Object or any subclass thereof.

XtParent returns the parent object for the specified widget. The returned object will be of class Object or a subclass.

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

w Specifies the widget. Must be of class Core or any subclass thereof.

XtScreen returns the screen pointer for the specified widget.

Window XtWindow(*w*)
  Widget *w*;

w Specifies the widget. Must be of class Core or any subclass thereof.

XtWindow returns the window of the specified widget.

The display pointer, screen pointer, and window of a widget or of the closest widget ancestor of a nonwidget object are available by means of **XtDisplayOfObject**, **XtScreenOfObject**, and **XtWindowOfObject**.

```
Display *XtDisplayOfObject(object)
      Widget object;
```

object Specifies the object. Must be of class Object or any subclass thereof.

XtDisplayOfObject is identical in function to **XtDisplay** if the object is a widget; otherwise **XtDisplayOfObject** returns the display pointer for the nearest ancestor of *object* that is of class Widget or a subclass thereof.

```
Screen *XtScreenOfObject(object)
      Widget object;
```

object Specifies the object. Must be of class Object or any subclass thereof.

XtScreenOfObject is identical in function to **XtScreen** if the object is a widget; otherwise **XtScreenOfObject** returns the screen pointer for the nearest ancestor of *object* that is of class Widget or a subclass thereof.

```
Window XtWindowOfObject(object)
      Widget object;
```

object Specifies the object. Must be of class Object or any subclass thereof.

XtWindowOfObject is identical in function to **XtWindow** if the object is a widget; otherwise **XtWindowOfObject** returns the window for the nearest ancestor of *object* that is of class Widget or a subclass thereof.

To retrieve the instance name of an object, use **XtName**.

```
String XtName(object)
      Widget object;
```

object Specifies the object whose name is desired. Must be of class Object or any subclass thereof.

XtName returns a pointer to the instance name of the specified object. The storage is owned by the Intrinsics and must not be modified. The name is not qualified by the names of any of the object's ancestors.

Several window attributes are locally cached in the widget instance. 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 Intrinsics. When an **XConfigureWindow** is issued by the Intrinsics on the widget's window (on request of its parent), these values are updated immediately rather than some time later when the server generates a **ConfigureNotify** event. (In fact, most widgets do not select **SubstructureNotify** events.) 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).

2.7.1. Unrealizing Widgets

To destroy the windows associated with a widget and its non-pop-up descendants, use **XtUnrealizeWidget**.

void XtUnrealizeWidget(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Core or any subclass thereof.

If the widget is currently unrealized, **XtUnrealizeWidget** simply returns. Otherwise it performs the following:

- Unmanages the widget if the widget is managed.
- Makes a postorder (child-to-parent) traversal of the widget tree rooted at the specified widget and, for each widget that has declared a callback list resource named ‘‘unrealizeCallback’’, executes the procedures on the XtNunrealizeCallback list.
- Destroys the widget’s window and any subwindows by calling **XDestroyWindow** with the specified widget’s *window* field.

Any events in the queue or which arrive following a call to **XtUnrealizeWidget** will be dispatched as if the window(s) of the unrealized widget(s) had never existed.

2.8. Destroying Widgets

The Intrinsics provide support

- To destroy all the pop-up children of the widget being destroyed and destroy all children of composite widgets.
- To remove (and unmap) the widget from its parent.
- To call the callback procedures that have been registered to trigger when the widget is destroyed.
- To minimize the number of things a widget has to deallocate when destroyed.
- To minimize the number of **XDestroyWindow** calls when destroying a widget tree.

To destroy a widget instance, use **XtDestroyWidget**.

void XtDestroyWidget(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Object or any subclass thereof.

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 1, **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 normal and pop-up 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, either normal or pop-up, of w1.

Phase 2 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 2, **XtDestroyWidget** performs the following on each entry in the destroy list in the order specified:

- Calls the destroy callback procedures registered on the widget and all normal and pop-up descendants in postorder (it calls child callbacks before parent callbacks).
- If the widget is not a pop-up child and the widget's parent is a subclass of **composite-WidgetClass**, 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 3.3).
- If the widget is not a pop-up child and the widget's parent is a subclass of **constraint-WidgetClass**, it calls the **ConstraintClassPart** destroy procedure for the parent, then for the parent's superclass, until finally it calls the **ConstraintClassPart** destroy procedure for **constraintWidgetClass**.
- Calls the destroy procedures for the widget and all normal and pop-up descendants in postorder. For each such widget, it calls the **CoreClassPart** 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 Object class record.
- Calls **XDestroyWindow** if the specified widget is realized (that is, has an X window). The server recursively destroys all normal descendant windows.
- Recursively descends the tree and destroys the windows for all realized pop-up descendants, deallocates all pop-up descendants, constraint records, callback lists, and if the widget's class is a subclass of **compositeWidgetClass**, children.

2.8.1. 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**; see Section 8.1.

2.8.2. Dynamic Data Deallocation: the destroy Procedure

The destroy procedure pointers in the **ObjectClassPart**, **RectObjClassPart**, and **CoreClassPart** structures are of type **XtWidgetProc**.

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

w Specifies the widget being destroyed.

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 ignore 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 passed 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 class record can be NULL.

Deallocating storage includes, but is not limited to, the following steps:

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

During destroy phase 2 for each widget, the Intrinsics remove the widget from the modal cascade, unregister all event handlers, remove all key, keyboard, button, and pointer grabs and remove all callback procedures registered on the widget. Any outstanding selection transfers will time out.

2.8.3. Dynamic Constraint Data Deallocation: the ConstraintClassPart 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 class of 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 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.

2.9. Exiting from an Application

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

Depending upon the widget set in use, it may be necessary to explicitly destroy individual widgets or widget trees with **XtDestroyWidget** before calling **XtDestroyApplicationContext** in order to ensure that any required widget cleanup is properly executed. The application developer must refer to the widget documentation to learn if a widget needs to perform additional cleanup beyond that performed automatically by the operating system. None of the widget classes defined by the Intrinsics require additional cleanup.

Chapter 3

Composite Widgets and Their Children

Composite widgets (widgets whose class is 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 Intrinsics 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 composite editor widget supporting multiple editing buffers might allocate one child widget for each file buffer, but it might only display a small number of the existing buffers. Widgets that are in this displayable subset are called managed widgets and enter into geometry manager calculations. The other children are called unmanaged widgets and, by definition, are not mapped by the Intrinsics.

Children are added to and removed from their parent's 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 declares 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 created by the parent. All these children are managed, and none ever makes 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. The class of heterogeneous boxes is usually a subclass of Constraint .

Shell boxes — Shell boxes typically have only one child, and the child's size is usually 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. 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);
    Widget w;
```

w Passes the newly created child.

Most composite widgets inherit their superclass's operation. The insert_child routine in **CompositeWidgetClass** calls and inserts the child at the specified position in the *children* list, expanding it if necessary.

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 class of their child widgets. A composite widget class that wishes to allow nonwidget children (see Chapter 12) must specify a **CompositeClassExtension** extension record as described in section 1.4.2.1 and set the *accepts_objects* field in this record to **True**. If the **CompositeClassExtension** record is not specified or the *accepts_objects* field is **False**, the composite widget can assume that all its children are of a subclass of Core without an explicit subclass test in the insert_child procedure.

If there is not enough room to insert a new child in the *children* array (that is, *num_children* is equal to *num_slots*), the insert_child procedure must first reallocate the array and update *num_slots*. The insert_child procedure then places the child at the appropriate position in the array and increments the *num_children* field.

3.2. Insertion Order of Children: the insert_position Procedure

Instances of composite widgets sometimes need to specify more 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 without constraining the order in which the buttons are created.

An application controls the presentation order of a set of children by supplying an XtNinsertPosition resource. The insert_position procedure pointer in a composite widget instance is of type **XtOrderProc**.

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

w Passes the newly created 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 using a composite class can apply different sorting criteria to widget instances of the class, passing in a different insert_position procedure resource 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.

3.3. Deletion of Children: the delete_child Procedure

To remove the child from the parent's *children* list, 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);
    Widget *w*;

w   Passes the child being deleted.

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.

3.4. Adding and Removing Children from the Managed Set

The Intrinsics 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 composite widget's change_managed procedure if the procedure pointer is non-NULL. The change_managed procedure pointer is of type **XtWidgetProc**. The widget argument specifies the composite widget whose managed child set has been modified.

3.4.1. Managing Children

To add a list of widgets to the geometry-managed (and hence displayable) subset of their Composite parent, use **XtManageChildren**.

typedef Widget *WidgetList;

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

children   Specifies a list of child widgets. Each child must be of class RectObj or any subclass thereof.

num_children   Specifies the number of children in the list.

The **XtManageChildren** function performs the following:

- Issues an error if the children do not all have the same parent or if the parent's class 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 which then calls **XMoveWindow**.

If the composite widget wishes to change the size or border width of any of its children, it calls **XtResizeWidget**, which first updates the *width*, *height*, and *border_width* fields and then calls **XConfigureWindow**. Simultaneous repositioning and resizing may be done with **XtConfigureWidget**; see Section 6.6.

To add a single child to its parent widget's set of managed children, use **XtManageChild**.

void XtManageChild(*child*)
    Widget *child*;

child Specifies the child. Must be of class RectObj or any subclass thereof.

The **XtManageChild** function constructs a **WidgetList** of length 1 and calls **XtManageChildren**.

To create and manage a child widget in a single procedure, use **XtCreateManagedWidget** or **XtVaCreateManagedWidget**.

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

name Specifies the resource instance name for the created widget.

widget_class Specifies the widget class pointer for the created widget. Must be **rectObjClass** or any subclass thereof.

parent Specifies the parent widget. Must be of class Composite or any subclass thereof.

args Specifies the argument list to override any other resource specifications.

num_args Specifies the number of entries in the argument list.

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

Widget XtVaCreateManagedWidget(*name*, *widget_class*, *parent*, ...)
    String *name*;
    WidgetClass *widget_class*;
    Widget *parent*;

name Specifies the resource instance name for the created widget.

widget_class Specifies the widget class pointer for the created widget. Must be **rectObjClass** or any subclass thereof.

parent Specifies the parent widget. Must be of class Composite or any subclass thereof.

... Specifies the variable argument list to override any other resource specifications.

XtVaCreateManagedWidget is identical in function to **XtCreateManagedWidget** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

3.4.2. 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. Each child must be of class RectObj or any subclass thereof.

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 child widgets. Removing widgets from a parent's managed set is often a temporary banishment, and some time later the client may manage the children again. To destroy widgets entirely, **XtDestroyWidget** should be called instead; see Section 2.9.

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

void XtUnmanageChild(*child*)
  Widget *child*;

child Specifies the child. Must be of class RectObj or any subclass thereof.

The **XtUnmanageChild** function constructs a widget list of length 1 and calls **XtUnmanageChildren**.

These 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.

3.4.3. 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. Must be of class Object or any subclass thereof.

The **XtIsManaged** function returns **True** if the specified widget is of class RectObj or any subclass thereof and is managed, or **False** otherwise.

3.5. 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. Must be of class Core or any subclass thereof.

map_when_managed
Specifies a Boolean value that indicates the new value that is stored into the widget's *map_when_managed* field.

If the widget is realized and managed and if *map_when_managed* is **True**, **XtSetMappedWhenManaged** maps the window. If the widget is realized and managed and if *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 XtNmappedWhenManaged resource then mapping the widget as appropriate. 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. Must be of class Core or any subclass thereof.

To unmap a widget explicitly, use **XtUnmapWidget**.

XtUnmapWidget(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Core or any subclass thereof.

3.6. Constrained Composite Widgets

The Constraint widget class is a subclass of **compositeWidgetClass**. The name is derived from the fact that constraint widgets 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 constraints as 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's class. 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 Intrinsics 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 does not belong to a subclass of **constraintWidgetClass**, then the child's *constraints* field is NULL.

Subclasses of Constraint can add constraint data to the constraint record defined by 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 class 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 class 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 fields 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 in the parent's class record to allocate a constraint record when a child is created. **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.

When the **XtGetValues** and **XtSetValues** functions are executed on a child, they use the constraint resources to get the values or set the values of constraints associated with that child. **XtSetValues** then calls the constraint set_values procedures so that the parent can recompute derived constraint fields and move or resize the child as appropriate. If a Constraint widget class or any of its superclasses have declared a **ConstraintClassExtension** record in the **ConstraintClassPart** *extension* fields with a record type of **NULLQUARK** and the *get_values_hook* field in the extension record is non-NULL, **XtGetValues** calls the get_values_hook procedure(s) to allow the parent to return derived constraint fields.

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.

Chapter 4

Shell Widgets

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 managed 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 have the **WM_TRANSIENT_FOR** property set. The effect of this property is dependent upon the window manager being used.
TopLevelShell	Used for normal top-level windows (for example, any additional top-level widgets an application needs).
ApplicationShell	Used for the single main top-level window that the window manager identifies as an application instance and that interacts with the session manager.

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 widget.

Shell widgets, whose class is a subclass of the Composite class, 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	The base class for shell widgets; provides the fields needed for all types of shells. Shell is a direct subclass of **compositeWidgetClass**.
OverrideShell	A subclass of Shell; used for shell windows that completely bypass the window manager.
WMShell	A subclass of Shell; contains fields needed by the common window manager protocol .
VendorShell	A subclass of WMShell; contains fields used by vendor-specific window managers.

TransientShell	A subclass of VendorShell; used for shell windows that desire the **WM_TRANSIENT_FOR** property.
TopLevelShell	A subclass of VendorShell; used for normal top level windows.
ApplicationShell	A subclass of TopLevelShell; used for an application's main top-level window.

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

4.1.1. ShellClassPart Definitions

Only the Shell class has additional class fields, which are all contained in the **ShellClassExtensionRec**. None of the other Shell classes have any additional class fields:

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

The full Shell class record definitions are

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

typedef struct {
        XtPointer next_extension;                      See Section 1.6.12
        XrmQuark record_type;                          See Section 1.6.12
        long version;                                  See Section 1.6.12
        Cardinal record_size;                          See Section 1.6.12
        XtGeometryHandler root_geometry_manager;       See below
} ShellClassExtensionRec, *ShellClassExtension;

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 single occurrences of the class records and pointers for creating instances of 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 whose class is a subclass of Shell.

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

The declarations for all Intrinsics-defined shells except VendorShell appear in **Shell.h** and **ShellP.h**. VendorShell has separate public and private .h files which are included by **Shell.h** and **ShellP.h**.

Shell.h uses incomplete structure definitions to ensure that the compiler catches attempts to access private data in any of the Shell instance or class data structures.

The symbolic constant for the **ShellClassExtension** version identifier is **XtShellExtensionVersion** (see Section 1.6.12).

The root_geometry_manager procedure acts as the parent geometry manager for geometry requests made by shell widgets. When a shell widget calls either **XtMakeGeometryRequest** or **XtMakeResizeRequest**, the root_geometry_manager procedure is invoked to negotiate the new geometry with the window manager. If the window manager permits the new geometry, the root_geometry_manager procedure should return **XtGeometryYes**; if the window manager denies the geometry request or it does not change the window geometry within some timeout interval (equal to *wm_timeout* in the case of WMShells), the root_geometry_manager procedure should return **XtGeometryNo**. If the window manager makes some alternative geometry change, the root_geometry_manager procedure may either return **XtGeometryNo** and handle the new geometry as a resize, or may return **XtGeometryAlmost** in anticipation that the shell will accept the compromise. If the compromise is not accepted, the new size must then be handled as a resize. Subclasses of Shell that wish to provide their own root_geometry_manager procedures are strongly encouraged to use enveloping to invoke their superclass's root_geometry_manager procedure under most situations, as the window manager interaction may be very complex.

If no **ShellClassPart** extension record is declared with *record_type* equal to **NULLQUARK**, then **XtInheritRootGeometryManager** is assumed.

4.1.2. ShellPart Definition

The various shell widgets have the following additional instance 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;
	Visual* visual;
} ShellPart;

typedef struct { int empty; } OverrideShellPart;

typedef struct {
	String title;
	int wm_timeout;
	Boolean wait_for_wm;
	Boolean transient;
	struct _OldXSizeHints {
		long flags;
		int x, y;
		int width, height;
		int min_width, min_height;
		int max_width, max_height;
		int width_inc, height_inc;
		struct {
			int x;
			int y;
		} min_aspect, max_aspect;
	} size_hints;
	XWMHints wm_hints;
	int base_width, base_height, win_gravity;
	Atom title_encoding;
} WMShellPart;

typedef struct {
	int vendor_specific;
} VendorShellPart;

typedef struct {
	Widget transient_for;
} TransientShellPart;

typedef struct {
	String icon_name;
	Boolean iconic;
	Atom icon_name_encoding;
} TopLevelShellPart;
```

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

The full shell widget instance record definitions are

```
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. Shell Resources

The resource names, classes, and representation types specified in the **shellClassRec** resource list are

Name	Class	Representation
XtNallowShellResize	XtCAllowShellResize	XtRBoolean
XtNcreatePopupChildProc	XtCCreatePopupChildProc	XtRFunction
XtNgeometry	XtCGeometry	XtRString
XtNoverrideRedirect	XtCOverrideRedirect	XtRBoolean
XtNpopdownCallback	XtCCallback	XtRCallback
XtNpopupCallback	XtCCallback	XtRCallback
XtNsaveUnder	XtCSaveUnder	XtRBoolean
XtNvisual	XtCVisual	XtRVisual

OverrideShell declares no additional resources beyond those defined by Shell.

The resource names, classes, and representation types specified in the **wmShellClassRec** resource list are

Name	Class	Representation
XtNbaseHeight	XtCBaseHeight	XtRInt
XtNbaseWidth	XtCBaseWidth	XtRInt
XtNheightInc	XtCHeightInc	XtRInt
XtNiconMask	XtCIconMask	XtRBitmap
XtNiconPixmap	XtCIconPixmap	XtRBitmap
XtNiconWindow	XtCIconWindow	XtRWindow
XtNiconX	XtCIconX	XtRInt
XtNiconY	XtCIconY	XtRInt
XtNinitialState	XtCInitialState	XtRInitialState
XtNinput	XtCInput	XtRBool
XtNmaxAspectX	XtCMaxAspectX	XtRInt
XtNmaxAspectY	XtCMaxAspectY	XtRInt
XtNmaxHeight	XtCMaxHeight	XtRInt
XtNmaxWidth	XtCMaxWidth	XtRInt
XtNminAspectX	XtCMinAspectX	XtRInt
XtNminAspectY	XtCMinAspectY	XtRInt
XtNminHeight	XtCMinHeight	XtRInt
XtNminWidth	XtCMinWidth	XtRInt
XtNtitle	XtCTitle	XtRString
XtNtitleEncoding	XtCTitleEncoding	XtRAtom

XtNtransient	XtCTransient	XtRBoolean
XtNwaitForWm	XtCWaitForWm	XtRBoolean
XtNwidthInc	XtCWidthInc	XtRInt
XtNwinGravity	XtCWinGravity	XtRInt
XtNwindowGroup	XtCWindowGroup	XtRWindow
XtNwmTimeout	XtCWmTimeout	XtRInt

The class resource list for VendorShell is implementation-defined.

The resource names, classes, and representation types that are specified in the **transient-ShellClassRec** resource list are

Name	Class	Representation
XtNtransientFor	XtCTransientFor	XtRWidget

The resource names, classes, and representation types that are specified in the **topLevelShellClassRec** resource list are

Name	Class	Representation
XtNiconName	XtCIconName	XtRString
XtNiconNameEncoding	XtCIconNameEncoding	XtRAtom
XtNiconic	XtCIconic	XtRBoolean

The resource names, classes, and representation types that are specified in the **application-ShellClassRec** resource list are

Name	Class	Representation
XtNargc	XtCArgc	XtRInt
XtNargv	XtCArgv	XtRStringArray

4.1.4. 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	(none)
spring_loaded	(none)
popped_up	**False**
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
visual	**CopyFromParent**

The *geometry* field specifies the size and position and is usually given only on a command line or in a defaults file. If the *geometry* field is non-NULL when a widget of class WMShell is realized, the geometry specification is parsed using **XWMGeometry** with a default geometry string constructed from the values of *x*, *y*, *width*, *height*, *width_inc*, and *height_inc* and the size and position flags in the window manager size hints are set. If the geometry specifies an x or y position, then **USPosition** is set. If the geometry specifies a width or height, then **USSize** is set. Any fields in the geometry specification override the corresponding values in the Core *x*, *y*, *width*, and *height* fields. If *geometry* is NULL or contains only a partial specification, then the Core *x*, *y*, *width*, and *height* fields are used and **PPosition** and **PSize** are set as appropriate. The geometry string is not copied by any of the Intrinsics Shell classes; a client specifying the string in an arglist or varargs list must ensure that the value remains valid until the shell widget is realized. For further information on the geometry string, see Section 10.3 in *Xlib – C Language X Interface*.

The *create_popup_child_proc* procedure is called by the **XtPopup** procedure and may remain NULL. The *grab_kind*, *spring_loaded*, and *popped_up* fields maintain widget state information as described under **XtPopup**, **XtMenuPopup**, **XtPopdown**, and **XtMenuPopdown**. The *allow_shell_resize* field controls whether the widget contained by the shell is allowed to try to resize itself. If allow_shell_resize is **False**, any geometry requests made by the child will always return **XtGeometryNo** without interacting with the window manager. Setting *save_under* **True** instructs the server to attempt to save the contents of windows obscured by the shell when it is mapped and to restore those contents automatically when the shell is unmapped. It is useful for pop-up menus. Setting *override_redirect* **True** determines whether the window manager can intercede when the shell window is mapped. The pop-up and pop-down callbacks are called during **XtPopup** and **XtPopdown**. For further information on override_redirect, see Section 3.2 in *Xlib – C Language X Interface* and Sections 4.1.10 and 4.2.2 in the *Inter-Client Communication Conventions Manual*.

The default values for Shell fields in WMShell and its subclasses are

Field	Default Value
title	Icon name, if specified, otherwise the application's name.
wm_timeout	Five seconds, in units of milliseconds.
wait_for_wm	**True**
transient	**True** for TransientShell, **False** otherwise
min_width	**XtUnspecifiedShellInt**
min_height	**XtUnspecifiedShellInt**
max_width	**XtUnspecifiedShellInt**
max_height	**XtUnspecifiedShellInt**
width_inc	**XtUnspecifiedShellInt**
height_inc	**XtUnspecifiedShellInt**
min_aspect_x	**XtUnspecifiedShellInt**
min_aspect_y	**XtUnspecifiedShellInt**
max_aspect_x	**XtUnspecifiedShellInt**
max_aspect_y	**XtUnspecifiedShellInt**
input	**False**
initial_state	Normal
icon_pixmap	None
icon_window	None
icon_x	**XtUnspecifiedShellInt**
icon_y	**XtUnspecifiedShellInt**
icon_mask	None

window_group	**XtUnspecifiedWindow**
base_width	**XtUnspecifiedShellInt**
base_height	**XtUnspecifiedShellInt**
win_gravity	**XtUnspecifiedShellInt**
title_encoding	See text

The *title* and *title_encoding* fields are stored in the **WM_NAME** property on the shell's window by the WMShell realize procedure. If the *title_encoding* field is **None**, the *title* string is assumed to be in the encoding of the current locale and the encoding of the **WM_NAME** property is set to **XStdICCTextStyle**. If a language procedure has not been set the default value of *title_encoding* is **XA_STRING**, otherwise the default value is **None**. The *wm_timeout* field specifies, in milliseconds, 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 **False** (later events may reset this value). When *wait_for_wm* is **False**, the shell does not wait for a response but relies on asynchronous notification. If *transient* is **True**, the **WM_TRANSIENT_FOR** property will be stored on the shell window with a value as specified below. The interpretation of this property is specific to the window manager under which the application is run; see the *Inter-Client Communication Conventions Manual* for more details. All other resources specify fields in the window manager hints and the window manager size hints. The realize and set_values procedures of WMShell set the corresponding flag bits in the hints if any of the fields contain non-default values. In addition, if a flag bit is set that refers to a field with the value **XtUnspecifiedShellInt**, the value of the field is modified as follows:

Field	Replacement
base_width, base_height	0
width_inc, height_inc	1
max_width, max_height	32767
min_width, min_height	1
min_aspect_x, min_aspect_y	-1
max_aspect_x, max_aspect_y	-1
icon_x, icon_y	-1
win_gravity	value returned by **XWMGeometry** if called, else **NorthWestGravity**

If the shell widget has a non-NULL parent, then the realize and set_values procedures replace the value **XtUnspecifiedWindow** in the *window_group* field with the window id of the root widget of the widget tree if the root widget is realized. The symbolic constant **XtUnspecifiedWindowGroup** may be used to indicate that the *window_group* hint flag bit is not to be set. If *transient* is **True** and the shell's class is not a subclass of TransientShell and *window_group* is not **XtUnspecifiedWindowGroup** the WMShell realize and set_values procedures then store the **WM_TRANSIENT_FOR** property with the value of *window_group*.

Transient shells have the following additional resource:

Field	Default Value
transient_for	NULL

The realize and set_values procedures of TransientShell store the **WM_TRANSIENT_FOR** property on the shell window if *transient* is **True**. If *transient_for* is non-NULL and the widget specified by *transient_for* is realized, then its window is used as the value of the **WM_TRANSIENT_FOR** property; otherwise, the value of *window_group* is used.

TopLevel shells have the the following additional resources:

Field	Default Value
icon_name	Shell widget's name
iconic	**False**
icon_name_encoding	See text

The *icon_name* and *icon_name_encoding* fields are stored in the **WM_ICON_NAME** property on the shell's window by the TopLevelShell realize procedure. If the *icon_name_encoding* field is **None**, the *icon_name* string is assumed to be in the encoding of the current locale and the encoding of the **WM_ICON_NAME** property is set to **XStdICCTextStyle**. If a language procedure has not been set the default value of *icon_name_encoding* is **XA_STRING**, otherwise the default value is **None**. The *iconic* field may be used by a client to request that the window manager iconify or deiconify the shell; the TopLevelShell set_values procedure will send the appropriate **WM_CHANGE_STATE** message (as specified by the *Inter-Client Communication Conventions Manual*) if this resource is changed from **False** to **True**, and will call **XtPopup** specifying *grab_kind* as **XtGrabNone** if *iconic* is changed from **True** to **False**. The XtNiconic resource is also an alternative way to set the XtNinitialState resource to indicate that a shell should be initially displayed as an icon; the TopLevelShell initialize procedure will set *initial_state* to **IconicState** if *iconic* is **True**.

Application shells have the following additional resources:

Field	Default Value
argc	0
argv	NULL

The *argc* and *argv* fields are used to initialize the standard property **WM_COMMAND**. See the *Inter-Client Communication Conventions Manual* for more information.

Chapter 5

Pop-Up Widgets

Pop-up widgets are used to create windows 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 normal widget children.

A parent of a pop-up widget does not actively manage its pop-up children; in fact, it usually does not operate upon them in any way. 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. The term *child* always refers to a normal, geometry-managed widget on the composite widget's list of children, and the term *pop-up child* always refers to a widget 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 dialog box that does not prevent continued interaction with the rest of the application) can usually be manipulated by the window manager and looks like any other application window from the user's point of view. The application main window itself is a special case of a modeless pop-up.

- Modal pop-ups

  A modal pop-up (for example, a dialog box that requires user input to continue) can sometimes be manipulated by the window manager, and except for events that occur in the dialog box, it disables user-event distribution to the rest of the application.

- Spring-loaded pop-ups

  A spring-loaded pop-up (for example, a menu) can seldom be manipulated by the window manager, and except for events that occur in the pop-up or its descendants, it disables user-event distribution to all other applications.

Modal pop-ups and spring-loaded pop-ups are very similar and should be coded as if they were the same. In fact, the same widget (for example, a ButtonBox or Menu widget) 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, all user input remap events occurring outside the spring-loaded pop-up (e.g., in a descendant) are also delivered to the spring-loaded pop-up after they have been dispatched to the appropriate descendant, so that, for example, buttond- up can take 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 allow user events to be dispatched to any of the modal or 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 one of the Shell widget classes.

5.2. Creating a Pop-Up Shell

For a widget to be popped up, it must be the child of a pop-up shell widget. None of the Intrinsics-supplied shells will simultaneously manage more than one 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 refer to the pop-up by the pop-up shell, not the 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 instance name for the created shell widget.

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

parent Specifies the parent widget. Must be of class Core or any subclass thereof.

args Specifies the argument list to override any other resource specifications.

num_args Specifies the number of entries 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 *popup_list* directly.

The screen resource for this widget is determined by first scanning *args* for the XtNscreen argument. If no XtNscreen argument is found, the resource database associated with the parent's screen is queried for the resource *name*.screen, class *Class*.Screen where *Class* is the *class_name* field from the **CoreClassPart** of the specified *widget_class* . If this query fails, the parent's screen is used. Once the screen is determined, the resource database associated with that screen is used to retrieve all remaining resources for the widget not specified in *args*.

A spring-loaded pop-up invoked from a translation table via **XtMenuPopup** must already 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 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.

To create a pop-up shell using varargs lists, use **XtVaCreatePopupShell**.

Widget XtVaCreatePopupShell(*name*, *widget_class*, *parent*, ...)
    String *name*;
    WidgetClass *widget_class*;
    Widget *parent*;

name Specifies the instance name for the created shell widget.

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

parent Specifies the parent widget. Must be of class Core or any subclass thereof.

... Specifies the variable argument list to override any other resource specifications.

XtVaCreatePopupShell is identical in function to **XtCreatePopupShell** with *the* args and *num_args* parameters replaced by a varargs list as described in Section 2.5.1.

5.3. Creating Pop-Up Children

Once a pop-up shell is created, the single child of the pop-up shell can be created either statically or dynamically.

At startup, an application can create the child of the pop-up shell, which is appropriate for pop-up children 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 might 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** or **XtPopupSpringLoaded**.
- One of the supplied callback procedures **XtCallbackNone**, **XtCallbackNonexclusive**, or **XtCallbackExclusive**.
- The standard translation action **XtMenuPopup**.

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

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

The create_popup_child_proc procedure pointer in the shell widget instance record is of type **XtCreatePopupChildProc**.

```
typedef void (*XtCreatePopupChildProc)(Widget);
      Widget w;
```

w Specifies the shell widget being popped up.

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 shell widget.

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

The **XtPopup** function performs the following:

- Calls **XtCheckSubclass** to ensure *popup_shell*'s class is a subclass of **shellWidgetClass**.

- Raises the window and returns if the shell's *popped_up* field is already **True**.
- Calls the callback procedures on the shell's *popup_callback* list, specifying a pointer to the value of *grab_kind* as the *call_data* argument.
- 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_proc* 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 **XMapRaised** with the window of *popup_shell*.

To map a spring-loaded pop-up from within an application, use **XtPopupSpringLoaded**.

void XtPopupSpringLoaded(*popup_shell*)
    Widget *popup_shell*;

popup_shell Specifies the shell widget to be popped up.

The **XtPopupSpringLoaded** function performs exactly as **XtPopup** except that it sets the shell *spring_loaded* field to **True** and always calls **XtAddGrab** with *exclusive* **True** and *spring-loaded* **True**.

To map a pop-up from a given widget's callback list, you also can register one of the **XtCallbackNone**, **XtCallbackNonexclusive**, or **XtCallbackExclusive** convenience routines as callbacks, using the pop-up shell widget as the client data.

void XtCallbackNone(*w*, *client_data*, *call_data*)
    Widget *w*;
    XtPointer *client_data*;
    XtPointer *call_data*;

w Specifies the widget.

client_data Specifies the pop-up shell.

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

void XtCallbackNonexclusive(*w*, *client_data*, *call_data*)
    Widget *w*;
    XtPointer *client_data*;
    XtPointer *call_data*;

w Specifies the widget.

client_data Specifies the pop-up shell.

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

void XtCallbackExclusive(*w*, *client_data*, *call_data*)
    Widget *w*;
    XtPointer *client_data*;
    XtPointer *call_data*;

w Specifies the widget.

client_data Specifies the pop-up shell.

call_data Specifies the callback data argument, 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 calling **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.

Within a translation table, to pop up a menu when a key or pointer button is pressed or when the pointer is moved into a widget, use **XtMenuPopup**, or its synonym, **MenuPopup**. From a translation writer's point of view, the definition for this translation action is

void XtMenuPopup(*shell_name*)
    String *shell_name*;

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

XtMenuPopup is known to the translation manager, which registers the corresponding built-in action procedure **XtMenuPopupAction** using **XtRegisterGrabAction** specifying *owner_events* **True**, *event_mask* **ButtonPressMask | ButtonReleaseMask**, and *pointer_mode* and *keyboard_mode* **GrabModeAsync**.

If **XtMenuPopup** is invoked on **ButtonPress**, it calls **XtPopupSpringLoaded** on the specified shell widget. If **XtMenuPopup** is invoked on **KeyPress** or **EnterWindow**, it calls **XtPopup** on the specified shell widget with *grab_kind* set to **XtGrabNonexclusive**. Otherwise, the translation manager generates a warning message and ignores the action.

XtMenuPopup tries to find the shell by searching the widget tree starting at the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that widget, it pops up the shell with the appropriate parameters. Otherwise, it moves up the parent chain to find a pop-up child with the specified name. If **XtMenuPopup** gets to the application top-level shell widget and has not found a matching shell, it generates a warning and returns immediately.

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 **XtMenuPopdown**

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

void XtPopdown(*popup_shell*)
    Widget *popup_shell*;

popup_shell Specifies the shell widget to pop down.

The **XtPopdown** function performs the following:

- Calls **XtCheckSubclass** to ensure *popup_shell*'s class is a subclass of **shellWidgetClass**.
- Checks that the *popped_up* field of *popup_shell* is **True**; otherwise, it returns immediately.
- Unmaps *popup_shell*'s window and, if *override_redirect* is **False**, sends a synthetic **UnmapNotify** event as specified by the *Inter-Client Communication Conventions Manual*.

- If *popup_shell*'s *grab_kind* is either **XtGrabNonexclusive** or **XtGrabExclusive**, it calls **XtRemoveGrab**.
- Sets *popup_shell*'s *popped_up* field to **False**.
- Calls the callback procedures on the shell's *popdown_callback* list, specifying a pointer to the value of the shell's *grab_kind* field as the *call_data* argument.

To pop down a pop-up from a callback list, you may use the callback **XtCallbackPopdown**.

void XtCallbackPopdown(*w*, *client_data*, *call_data*)
  Widget *w*;
  XtPointer *client_data*;
  XtPointer *call_data*;

w Specifies the widget.

client_data Specifies a pointer to the **XtPopdownID** structure.

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

The **XtCallbackPopdown** function casts the *client_data* parameter to a pointer of type **XtPopdownID**.

```
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 usually the widget that was used to pop it up in one of the pop-up callback convenience procedures.

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

Within a translation table, to pop down a spring-loaded menu when a key or pointer button is released or when the pointer is moved into a widget, use **XtMenuPopdown** or its synonym, **MenuPopdown**. From a translation writer's point of view, the definition for this translation action is

void XtMenuPopdown(*shell_name*)
  String *shell_name*;

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

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

Chapter 6

Geometry Management

A widget does not directly control its size and location; rather, its parent is responsible for controlling them. 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 specified in the *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 is 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 each 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**. 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 allow the request, disallow the request, or 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 the width, height, or border width) other than the one making the request, it should do so by calling **XtResizeWidget**. The requesting child may be given special treatment; see Section 6.5. 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, the composite widget may want 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; see Sections 2.5 and 2.6.

Notes

1. The Intrinsics 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 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

When making a geometry request, the child specifies an **XtWidgetGeometry** 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;
```

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*;

w Specifies the widget making the request. Must be of class RectObj or any subclass thereof.

request Specifies the desired widget geometry (size, position, border width, and stacking order).

reply_return Returns the allowed widget size, or may be NULL if the requesting widget is not interested in handling **XtGeometryAlmost**.

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's class is not a subclass of **compositeWidgetClass** or the parent's *geometry_manager* field 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 *request->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** just returns the resulting value from the parent's geometry manager.

Children of primitive widgets are always unmanaged; therefore, **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 *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 Intrinsics 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 the caller wants 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 Intrinsics also support the following value.

#define	**XtSMDontChange**	5

For definition and behavior of **Above**, **Below**, **TopIf**, **BottomIf**, and **Opposite**, see Section 3.7 in *Xlib – C Language X Interface*. **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 making the request. Must be of class RectObj or any subclass thereof.

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 an **XtWidgetGeometry** structure and specifies that width and height should change by setting *request_mode* to **CWWidth | CWHeight**. 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 call **XtMakeResizeRequest** again 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 original 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 1 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*;

w	Passes the widget making the request.
request	Passes the new geometry the child desires.
geometry_return	Passes a geometry structure in which the geometry manager may store a compromise.

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

A bit set to zero in the request's *request_mode* field means that the child widget does not care about the value of the corresponding field, so the geometry manager can change this field 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* fields appropriately. Then, it returns **XtGeometryYes**, and the values pointed to by the *geometry_return* argument are undefined. The widget's window is moved and resized automatically by **XtMakeGeometry-Request**.

Homogeneous composite widgets often find it convenient to treat the widget making the request the same as any other widget, including reconfiguring it using **XtConfigureWidget** or **XtResizeWidget** 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.

Note

> To remain compatible with layout techniques used in older widgets (before **XtGeometryDone** was added to the Intrinsics), a geometry manager should avoid using **XtResizeWidget** or **XtConfigureWidget** on the child making the request because the layout process of the child may be in an intermediate state in which it is not prepared to handle a call to its resize procedure. A self-contained widget set may choose this alternative geometry management scheme, however, provided that it clearly warns widget developers of the compatibility consequences.

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 disallows the request, the widget cannot change its geometry. The values pointed to by *geometry_return* are undefined, and the geometry manager returns **XtGeometryNo**.

Sometimes the geometry manager cannot satisfy the request exactly but 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 the structure pointed to by *geometry_return* with the actual changes it is willing to make, including an appropriate *request_mode* mask, and returns **XtGeometryAlmost**. If a bit in *geometry_return*->*request_mode* is zero, the geometry manager agrees not to change the corresponding value if *geometry_return* is used immediately in a new request. If a bit is 1, the geometry manager does change that element to the corresponding value in *geometry_return*. More bits may be set in *geometry_return*->*request_mode* 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* values filled in by the geometry manager with 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, dynamic behavior in the user's window manager may affect the final outcome.

To return **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 data describing 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, the parent uses **XtMoveWidget**.

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

w Specifies the widget. Must be of class RectObj or any subclass thereof.

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 object and, if the object is a widget and is realized, issues an Xlib **XMoveWindow** call on the widget's window.

To resize a sibling widget of the child making the geometry request, the parent uses **XtResizeWidget**.

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

w Specifies the widget. Must be of class RectObj or any subclass thereof.

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 object and, if the object is a widget and is realized, issues an **XConfigureWindow** call on the widget's window.

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

To move and resize the sibling widget of the child making the geometry request, the parent uses **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. Must be of class RectObj or any subclass thereof.

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 new geometry fields are all equal to the current values. Otherwise, **XtConfigureWidget** writes the new *x*, *y*, *width*, *height*, and *border_width* values into the object and, if the object is a widget and is realized, makes an Xlib **XConfigureWindow** call on the widget's window.

If the new width or height is different from its old value, **XtConfigureWidget** calls the object'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, the parent uses **XtResizeWindow**.

void XtResizeWindow(*w*)
    Widget *w*;

w Specifies the widget. Must be of class Core or any subclass thereof.

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 inexpensive 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, the parent 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. Must be of class RectObj or any subclass thereof.

intended Specifies the new geometry the parent plans to give to the child, or NULL.

preferred_return Returns the child widget's preferred geometry.

To discover a child's preferred geometry, the child's parent stores the new geometry in the corresponding fields of the intended structure, sets the corresponding bits in *intended.request_mode*, and calls **XtQueryGeometry**. The parent should set only those fields that are important to it so that the child can determine whether it may be able to attempt changes to other fields.

XtQueryGeometry clears all bits in the *preferred_return->request_mode* field 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* equal to zero before calling the query_geometry procedure.

Note

> If **XtQueryGeometry** is called from within a geometry_manager procedure for the widget that issued **XtMakeGeometryRequest** or **XtMakeResizeRequest**, the results are not guaranteed to be consistent with the requested changes. The change request passed to the geometry manager takes precedence over the preferred geometry.

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

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

w	Passes the child widget whose preferred geometry is required.
request	Passes the geometry changes which the parent plans to make.
preferred_return	Passes a structure in which the child returns its preferred geometry.

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 *preferred_return* (setting the bits in *preferred_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 *preferred_return* with a bit set in *preferred_return->request_mode* is different from the corresponding field in *request* or if a bit was set in *preferred_return->request_mode* that was not set in the 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**.

Note

> The query_geometry procedure may assume that no **XtMakeResizeRequest** or **XtMakeGeometryRequest** is in progress for the specified widget; that is, it is not required to construct a reply consistent with the requested geometry if such a request were actually outstanding.

After calling the query_geometry procedure or if the *query_geometry* field is NULL, **XtQuery-Geometry** examines all the unset bits in *preferred_return->request_mode* and sets the corresponding fields in *preferred_return* to the current values from the widget instance. If **CWStackMode** is not set, the *stack_mode* field is set to **XtSMDontChange**. **XtQuery-Geometry** 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 the reply and, more important, 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 in the widget instance is its preferred value. In addition, whether or not the caller ignores the return value or the reply mask, it is guaranteed that the *preferred_return* structure contains complete geometry information for the child.

Parents are expected to call **XtQueryGeometry** in their layout routine and wherever else the information is significant after change_managed has been called. The first time it is invoked, 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.

Chapter 7

Event Management

While Xlib allows the reading and processing of events anywhere in an application, widgets in the X 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.
- Register procedures to be called when specific events arrive.

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 Intrinsics 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 context, use **XtAppAddInput**.

XtInputId XtAppAddInput(*app_context*, *source*, *condition*, *proc*, *client_data*)
    XtAppContext *app_context*;
    int *source*;
    XtPointer *condition*;
    XtInputCallbackProc *proc*;
    XtPointer *client_data*;

app_context Specifies the application context that identifies the application.

source Specifies the source file descriptor on a POSIX-based system or other operating-system-dependent device specification.

condition Specifies the mask that indicates a read, write, or exception condition or some other operating-system-dependent condition.

proc Specifies the procedure to be called when the condition is found.

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

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 an event is pending on this source, the callback procedure is called.

The legal values for the *condition* argument are operating-system-dependent. On a POSIX-based system, *source* is a file number and the condition is some union of the following:

XtInputReadMask Specifies that *proc* is to be called when *source* has data to be read.

XtInputWriteMask Specifies that *proc* is to be called when *source* is ready for writing.

XtInputExceptMask Specifies that *proc* is to be called when *source* has exception data.

Callback procedure pointers used to handle file events are of type **XtInputCallbackProc**.

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

client_data Passes the client data argument that was registered for this procedure in **XtAppAddInput**.

source Passes the source file descriptor generating the event.

id Passes 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 events from the file source specified by *id*.

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 register a timeout callback, use **XtAppAddTimeOut**.

XtIntervalId XtAppAddTimeOut(*app_context*, *interval*, *proc*, *client_data*)
    XtAppContext *app_context*;
    unsigned long *interval*;
    XtTimerCallbackProc *proc*;
    XtPointer *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 to be called when the time expires.

client_data Specifies an argument 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 *proc* is called when **XtAppNextEvent** or **XtAppProcessEvent** is next called after the time interval elapses, and then the timeout is removed.

Callback procedure pointers used with timeouts are of type **XtTimerCallbackProc**.

typedef void (*XtTimerCallbackProc)(XtPointer, XtIntervalId*);
    XtPointer *client_data*;
    XtIntervalId **timer*;

client_data Passes the client data argument that was registered for this procedure in **XtAppAddTimeOut**.

timer Passes 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 cleared.

The **XtRemoveTimeOut** function removes the pending 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 directed 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 popup 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 exposure events and the like keep the application's display up-todate. Any event that occurs within the cascade is delivered as usual. The user events 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 ignored when they occur outside the cascade are **MotionNotify** and **EnterNotify**. All other events are delivered normally. In particular, note that this is one way in which widgets can receive **LeaveNotify** events without first receiving **EnterNotify** events; they should be prepared to deal with this, typically by ignoring any unmatched **LeaveNotify** events.

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 is popped down.

To constrain or 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. Must be of class Core or any subclass thereof.

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 to the modal cascade and checks that *exclusive* is **True** if *spring_loaded* is **True**. If this condition is not met, **XtAddGrab** generates a warning message.

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 in the application they occur, remap events are always delivered to the most recent widget in the active subset of the cascade registered with *spring_loaded* **True**, if any such widget exists. If the event occurred in the active subset of the cascade but outside the spring-loaded widget, it is delivered normally before being delivered also to the spring-loaded widget. Regardless of where it is dispatched, the Intrinsics do not modify the contents of the event.

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 a warning if the specified widget is not on the modal cascade.

7.2.1. Requesting Key and Button Grabs

The Intrinsics provide a set of key and button grab interfaces that are parallel to those provided by Xlib and that allow the Intrinsics to modify event dispatching when necessary. X Toolkit applications and widgets that need to passively grab keys or buttons or actively grab the keyboard or pointer should use the following Intrinsics routines rather than the corresponding Xlib routines.

To passively grab a single key of the keyboard, use **XtGrabKey**.

void XtGrabKey(*widget*, *keycode*, *modifiers*, *owner_events*, *pointer_mode*, *keyboard_mode*)
  Widget *widget*;
  KeyCode *keycode*;
  Modifiers *modifiers*;
  Boolean *owner_events*;
  int *pointer_mode*, *keyboard_mode*;

widget Specifies the widget in whose window the key is to be grabbed. Must be of class Core or any subclass thereof.

keycode
modifiers
owner_events
pointer_mode
keyboard_mode Specify arguments to **XGrabKey**; see Section 12.2 in *Xlib – C Language X Interface*.

XtGrabKey calls **XGrabKey** specifying the widget's window as the grab window if the widget is realized. The remaining arguments are exactly as for **XGrabKey**. If the widget is not realized, or is later unrealized, the call to **XGrabKey** will be performed (again) when the widget is realized and its window becomes mapped. In the future, if **XtDispatchEvent** is called with a **KeyPress** event matching the specified keycode and modifiers (which may be **AnyKey** or **AnyModifier**, respectively) for the widget's window, the Intrinsics will call **XtUngrabKeyboard** with the timestamp from the **KeyPress** event if either of the following conditions is true:

- There is a modal cascade and the widget is not in the active subset of the cascade and the keyboard was not previously grabbed, or
- **XFilterEvent** returns **True**.

To cancel a passive key grab, use **XtUngrabKey**.

void XtUngrabKey(*widget*, *keycode*, *modifiers*)
    Widget *widget*;
    KeyCode *keycode*;
    Modifiers *modifiers*;

widget Specifies the widget in whose window the key was grabbed.

keycode
modifiers Specify arguments to **XUngrabKey**; see Section 12.2 in *Xlib – C Language X Interface*.

The **XtUngrabKey** procedure calls **XUngrabKey** specifying the widget's window as the ungrab window if the widget is realized. The remaining arguments are exactly as for **XUngrabKey**. If the widget is not realized, **XtUngrabKey** removes a deferred **XtGrabKey** request, if any, for the specified widget, keycode, and modifiers.

To actively grab the keyboard, use **XtGrabKeyboard**.

int XtGrabKeyboard(*widget*, *owner_events*, *pointer_mode*, *keyboard_mode*, *time*)
    Widget *widget*;
    Boolean *owner_events*;
    int *pointer_mode*, *keyboard_mode*;
    Time *time*;

widget Specifies the widget for whose window the keyboard is to be grabbed. Must be of class Core or any subclass thereof.

owner_events
pointer_mode
keyboard_mode
time Specify arguments to **XGrabKeyboard**; see Section 12.2 in *Xlib – C Language X Interface*.

If the specified widget is realized **XtGrabKeyboard** calls **XGrabKeyboard** specifying the widget's window as the grab window. The remaining arguments and return value are exactly as for **XGrabKeyboard**. If the widget is not realized, **XGrabKeyboard** immediately returns

GrabNotViewable. No future automatic ungrab is implied by **XtGrabKeyboard**.

To cancel an active keyboard grab, use **XtUngrabKeyboard**.

void XtUngrabKeyboard(*widget*, *time*)
    Widget *widget*;
    Time *time*;

widget Specifies the widget that has the active keyboard grab.

time Specifies the additional argument to **XUngrabKeyboard**; see Section 12.2 in *Xlib – C Language X Interface*.

XtUngrabKeyboard calls **XUngrabKeyboard** with the specified time.

To passively grab a single pointer button, use **XtGrabButton**.

void XtGrabButton(*widget*, *button*, *modifiers*, *owner_events*, *event_mask*, *pointer_mode*,
        keyboard_mode, *confine_to*, *cursor*)
    Widget *widget*;
    int *button*;
    Modifiers *modifiers*;
    Boolean *owner_events*;
    unsigned int *event_mask*;
    int *pointer_mode*, *keyboard_mode*;
    Window *confine_to*;
    Cursor *cursor*;

widget Specifies the widget in whose window the button is to be grabbed. Must be of class Core or any subclass thereof.

button
modifiers
owner_events
event_mask
pointer_mode
keyboard_mode
confine_to
cursor Specify arguments to **XGrabButton**; see Section 12.1 in *Xlib – C Language X Interface*.

XtGrabButton calls **XGrabButton** specifying the widget's window as the grab window if the widget is realized. The remaining arguments are exactly as for **XGrabButton**. If the widget is not realized, or is later unrealized, the call to **XGrabButton** will be performed (again) when the widget is realized and its window becomes mapped. In the future, if **XtDispatchEvent** is called with a **ButtonPress** event matching the specified button and modifiers (which may be **AnyButton** or **AnyModifier**, respectively) for the widget's window, the Intrinsics will call **XtUngrabPointer** with the timestamp from the **ButtonPress** event if either of the following conditions is true:

- There is a modal cascade and the widget is not in the active subset of the cascade and the pointer was not previously grabbed, or
- **XFilterEvent** returns **True**.

To cancel a passive button grab, use **XtUngrabButton**.

void XtUngrabButton(*widget*, *button*, *modifiers*)
    Widget *widget*;
    unsigned int *button*;
    Modifiers *modifiers*;

widget Specifies the widget in whose window the button was grabbed.

button
modifiers Specify arguments to **XUngrabButton**; see Section 12.1 in *Xlib – C Language X Interface*.

The **XtUngrabButton** procedure calls **XUngrabButton** specifying the widget's window as the ungrab window if the widget is realized. The remaining arguments are exactly as for **XUngrabButton**. If the widget is not realized, **XtUngrabButton** removes a deferred **XtGrabButton** request, if any, for the specified widget, button, and modifiers.

To actively grab the pointer, use **XtGrabPointer**.

int XtGrabPointer(*widget*, *owner_events*, *event_mask*, *pointer_mode*, *keyboard_mode*,
        confine_to, *cursor*, *time*)
    Widget *widget*;
    Boolean *owner_events*;
    unsigned int *event_mask*;
    int *pointer_mode*, *keyboard_mode*;
    Window *confine_to*;
    Cursor *cursor*;
    Time *time*;

widget Specifies the widget for whose window the pointer is to be grabbed. Must be of class Core or any subclass thereof.

owner_events
event_mask
pointer_mode
keyboard_mode
confine_to
cursor
time Specify arguments to **XGrabPointer**; see Section 12.1 in *Xlib – C Language X Interface*.

If the specified widget is realized, **XtGrabPointer** calls **XGrabPointer**, specifying the widget's window as the grab window. The remaining arguments and return value are exactly as for **XGrabPointer**. If the widget is not realized, **XGrabPointer** immediately returns **GrabNotViewable**. No future automatic ungrab is implied by **XtGrabPointer**.

To cancel an active pointer grab, use **XtUngrabPointer**.

void XtUngrabPointer(*widget*, *time*)
    Widget *widget*;
    Time *time*;

widget Specifies the widget that has the active pointer grab.

time Specifies the time argument to **XUngrabPointer**; see Section 12.1 in *Xlib – C Language X Interface*.

XtUngrabPointer calls **XUngrabPointer** with the specified time.

7.3. Focusing Events on a Child

To redirect keyboard input to a normal descendant of a widget without calling **XSetInputFocus**, use **XtSetKeyboardFocus**.

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

subtree Specifies the subtree of the hierarchy for which the keyboard focus is to be set. Must be of class Core or any subclass thereof.

descendant Specifies either the normal (non-pop-up) descendant of *subtree* to which keyboard events are logically directed, or **None**. It is not an error to specify **None** when no input focus was previously set. Must be of class Object or any subclass thereof.

XtSetKeyboardFocus causes **XtDispatchEvent** to remap keyboard events occurring within the specified subtree and dispatch them to the specified descendant widget or to an ancestor. If the descendant's class is not a subclass of Core, the descendant is replaced by its closest windowed ancestor.

When there is no modal cascade, keyboard events can be dispatched to a widget in one of five ways. Assume the server delivered the event to the window for widget E (because of X input focus, key or keyboard grabs, or pointer position).

- If neither E nor any of E's ancestors have redirected the keyboard focus, or if the event activated a grab for E as specified by a call to **XtGrabKey** with any value of *owner_events*, or if the keyboard is actively grabbed by E with *owner_events* **False** via **XtGrabKeyboard** or **XtGrabKey** on a previous key press, the event is dispatched to E.
- Beginning with the ancestor of E closest to the root that has redirected the keyboard focus or E if no such ancestor exists, if the target of that focus redirection has in turn redirected the keyboard focus, recursively follow this focus chain to find a widget F that has not redirected focus.
  - If E is the final focus target widget F or a descendant of F, the event is dispatched to E.
  - If E is not F, an ancestor of F, or a descendant of F, and the event activated a grab for E as specified by a call to **XtGrabKey** for E, **XtUngrabKeyboard** is called.
  - If E is an ancestor of F, and the event is a key press, and either
    + E has grabbed the key with **XtGrabKey** and *owner_events* **False**, or
    + E has grabbed the key with **XtGrabKey** and *owner_events* **True**, and the coordinates of the event are outside the rectangle specified by E's geometry,

    then the event is dispatched to E.
  - Otherwise, define A as the closest common ancestor of E and F:
    + If there is an active keyboard grab for any widget via either **XtGrabKeyboard** or **XtGrabKey** on a previous key press, or if no widget between F and A (noninclusive) has grabbed the key and modifier combination with **XtGrabKey** and any value of *owner_events*, the event is dispatched to F.
    + Else, the event is dispatched to the ancestor of F closest to A that has grabbed the key and modifier combination with **XtGrabKey**.

When there is a modal cascade, if the final destination widget as identified above is in the active subset of the cascade, the event is dispatched; otherwise the event is remapped to a spring-loaded shell or discarded. Regardless of where it is dispatched, the Intrinsics do not modify the contents of the event.

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 the subtree, a **FocusIn** event is generated for the descendant if **FocusChange** events have been selected by the descendant. Similarly, when *subtree* loses the X input focus or the keyboard focus for one of its ancestors, a

FocusOut event is generated for descendant if **FocusChange** events have been selected by the descendant.

A widget tree may also actively manage the X server input focus. To do so, a widget class specifies an accept_focus procedure.

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, pursuant to the restrictions of the *Inter-Client Communication Conventions Manual*. To allow outside agents, such as the parent, to cause a widget to take the input focus, every widget exports an accept_focus procedure. The widget returns a value indicating 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 classes that never want the input focus should set the *accept_focus* field to NULL.

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

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

w Specifies the widget. Must be of class Core or any subclass thereof.

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

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 are Intrinsics 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 buffers of each Display in the application context and returns zero.

To return the event 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 X event in the queue, **XtAppPeekEvent** copies it into *event_return* and returns **True**. If no X input is on the queue, **XtAppPeekEvent** flushes the output buffers of each Display in the application context and blocks until some input is available (possibly calling some timeout callbacks in the interim). If the next available input is an X event, **XtAppPeekEvent** fills in *event_return* and returns **True**. Otherwise, the input is for an input source registered with **XtAppAddInput**, and **XtAppPeekEvent** returns **False**.

To remove and return the event from the head of a given application's X event 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 the X event queue is empty, **XtAppNextEvent** flushes the X output buffers of each Display in the application context and waits for an X 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) 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, **Intrinsic.h** defines the symbolic name **XtIMAll** to be the bitwise inclusive OR of these three event types.

The **XtAppProcessEvent** function processes one timer, input source, or X event. If there is no event or input of the appropriate type to process, then **XtAppProcessEvent** blocks until there is. If there is more than one type of input 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, input sources by calling any appropriate 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 no handlers for that event are registered, the event is ignored and the dispatcher simply returns.

To dispatch an event returned by **XtAppNextEvent**, retrieved directly from the Xlib queue, or synthetically constructed, to any registered event filters or event handlers call **XtDispatchEvent**.

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

event Specifies a pointer to the event structure to be dispatched to the appropriate event handlers.

The **XtDispatchEvent** function first calls **XFilterEvent** with the *event* and the window of the widget to which the Intrinsics intend to dispatch the event, or the event window if the Intrinsics would not dispatch the event to any handlers. If **XFilterEvent** returns **True** and the event activated a server grab as identified by a previous call to **XtGrabKey** or **XtGrabButton**, **XtDispatchEvent** calls **XtUngrabKeyboard** or **XtUngrabPointer** with the timestamp from the event and immediately returns **True**. If **XFilterEvent** returns **True** and a grab was not activated, **XtDispatchEvent** just immediately returns **True**. Otherwise, **XtDispatchEvent** sends the event to the event handler functions that have been previously registered with the dispatch routine. **XtDispatchEvent** returns **True** if **XFilterEvent** returned **True**, or if the event was dispatched to some handler and **False** if it found no handler to which to dispatch the event. **XtDispatchEvent** records the last timestamp in any event that contains a timestamp (see **XtLastTimestampProcessed**), regardless of whether it was filtered or dispatched. If a modal cascade is active with *spring_loaded* **True**, and if the event is a remap event as defined by **XtAddGrab**, **XtDispatchEvent** may dispatch the event a second time. If it does so, **XtDispatchEvent** will call **XFilterEvent** again with the window of the spring-loaded widget prior to the second dispatch and if **XFilterEvent** returns **True**, the second dispatch will not be performed.

7.6. The Application Input Loop

To process all input from a given application in a continuous loop, use the convenience procedure **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 dispatches the event to the appropriate registered procedure by calling **XtDispatchEvent**. This constitutes the main loop of X Toolkit applications, and, as such, it does not return. Applications are expected to exit in response to some user action within a callback procedure. 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 grayed 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 ancestors 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.

Note

> Pop-up shells will have *ancestor_sensitive* **False** if the parent was insensitive when the shell was created. Since **XtSetSensitive** on the parent will not modify the resource of the pop-up child, clients are advised to include a resource specification of the form ‘‘*TransientShell.ancestorSensitive: True’’ in the application defaults resource file or to otherwise ensure that the parent is sensitive when creating pop-up shells.

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

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

w Specifies the widget. Must be of class RectObj or any subclass thereof.

sensitive Specifies whether the widget should receive keyboard, pointer, and focus events.

The **XtSetSensitive** function first calls **XtSetValues** on the current widget with an argument list specifying the XtNsensitive resource and the new value. If *sensitive* is **False** and the widget's class is a subclass of Composite, **XtSetSensitive** recursively propagates the new value down the child tree by calling **XtSetValues** on each child to set *ancestor_sensitive* to **False**. If *sensitive* is **True** and the widget's class is a subclass of Composite and the widget's *ancestor_sensitive* field is **True**, **XtSetSensitive** sets the *ancestor_sensitive* of each child to **True** and then recursively calls **XtSetValues** on each normal descendant that is now sensitive to set *ancestor_sensitive* to **True**.

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

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

To check the current sensitivity state of a widget, use **XtIsSensitive**.

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

w Specifies the object. Must be of class Object or any subclass thereof.

The **XtIsSensitive** function returns **True** or **False** to indicate whether user input events are being dispatched. If object's class is a subclass of RectObj and both *sensitive* and *ancestor_sensitive* are **True**, **XtIsSensitive** returns **True**; otherwise, it returns **False**.

7.8. Adding Background Work Procedures

The Intrinsics have some 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)(XtPointer);
  XtPointer *client_data*;

client_data Passes the client data specified when the work procedure was registered.

This procedure should return **True** when it is done to indicate that it should be removed. If the procedure returns **False**, it will remain registered and will be called again when the application is next idle. Work procedures should be very judicious about how much they do. If they run for more than a small part of a second, interactive feel 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*;
  XtPointer *client_data*;

app_context Specifies the application context that identifies the application.

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

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

The **XtAppAddWorkProc** function adds the specified work procedure for the application identified by *app_context* and returns 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 specific X events. The filters, which screen out events that are redundant or are temporarily unwanted, handle pointer motion compression, enter/leave compression, and exposure compression.

7.9.1. Pointer Motion Compression

Widgets can have a hard time keeping up with a rapid stream of pointer motion events. Further, they usually do not 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 for the same widget 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 exposure events. The *compress_exposure* field in the widget class structure specifies the type and number of exposure events that will be dispatched to the widget's expose procedure. This field must be initialized to one of the following values,

```
#define XtExposeNoCompress          ((XtEnum)False)
#define XtExposeCompressSeries      ((XtEnum)True)
#define XtExposeCompressMultiple    <implementation-defined>
#define XtExposeCompressMaximal     <implementation-defined>
```

optionally ORed with any combination of the following flags (all with implementation-defined values):

XtExposeGraphicsExpose, **XtExposeGraphicsExposeMerged** and **XtExposeNoExpose**.

If the *compress_exposure* field in the widget class structure does not specify **XtExposeNoCompress**, the event manager calls the widget's expose procedure only once for a series of exposure events. In this case, all **Expose** or **GraphicsExpose** events are accumulated into a region. When the final event 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. For more information on regions, see Section 16.5 in *Xlib – C Language X Interface*.)

The values have the following interpretation:

XtExposeNoCompress

No exposure compression is performed; every selected event is individually dispatched to the expose procedure with a *region* argument of NULL.

XtExposeCompressSeries

Each series of exposure events is coalesced into a single event, which is dispatched when an exposure event with count equal to zero is reached.

XtExposeCompressMultiple

Consecutive series of exposure events are coalesced into a single event, which is dispatched when an exposure event with count equal to zero is reached and either the event queue is empty or the next event is not an exposure event for the same widget.

XtExposeCompressMaximal

All expose series currently in the queue for the widget are coalesced into a single event without regard to intervening nonexposure events. If a partial series is in the end of the queue, the Intrinsics will block until the end of the series is received.

The additional flags have the following meaning:

XtExposeGraphicsExpose

Specifies that **GraphicsExpose** events are also to be dispatched to the expose procedure. **GraphicsExpose** events will be compressed, if specified, in the same manner as **Expose** events.

XtExposeGraphicsExposeMerged

Specifies in the case of **XtExposeCompressMultiple** and **XtExposeCompressMaximal** that series of **GraphicsExpose** and **Expose** events are to be compressed together, with the final event type determining the type of the event passed to the expose procedure. If this flag is not set, then only series of the same event type as the event at the head of the queue are coalesced. This flag also implies **XtExposeGraphicsExpose**.

XtExposeNoExpose

Specifies that **NoExpose** events are also to be dispatched to the expose procedure. **NoExpose** events are never coalesced with other exposure events or with each other.

7.10. Widget Exposure and Visibility

Every primitive widget and some composite widgets display data on the screen by means of direct 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 specifies **XtExposeNoCompress** or the event type is **NoExpose** (see Section 7.9.3), *region* is NULL; otherwise, the event is the final event in the compressed series but *x*, *y*, *width*, and *height* contain the bounding box for *region*. The region is created and destroyed by the Intrinsics, but the widget is permitted to modify the region contents.

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 not chained and is therefore 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 implement separate display procedures for the widgets Label, Pushbutton, and Toggle, you could write a single display routine in Label that uses display state fields like

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

Label would have *invert* and *highlight* always **False** and *highlight_width* zero. Pushbutton 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 Pushbutton and Toggle inherit their superclass's expose procedure; see Section 1.6.10.

7.10.2. Widget Visibility

Some widgets may use substantial computing resources to produce the data they will display. 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 compute display data. This field is guaranteed to be **True** by the time an exposure event is processed if any part of the widget is visible but is **False** if the widget is fully obscured.

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 sets *visible* to **True** on **VisibilityUnobscured** or **VisibilityPartiallyObscured** events and **False** on **VisibilityFullyObscured** events.

7.11. X Event Handlers

Event handlers are procedures 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, XtPointer, XEvent*, Boolean*);
    Widget *w*;
    XtPointer *client_data*;
    XEvent **event*;
    Boolean **continue_to_dispatch*;

w Specifies the widget for which the event arrived.

client_data Specifies any client-specific information registered with the event handler.

event Specifies the triggering event.

continue_to_dispatch
Specifies whether the remaining event handlers registered for the current event should be called.

After receiving an event and before calling any event handlers, the Boolean pointed to by *continue_to_dispatch* is initialized to **True**. When an event handler is called, it may decide that further processing of the event is not desirable and may store **False** in this Boolean, in which case any handlers remaining to be called for the event will be ignored.

The circumstances under which the Intrinsics may add event handlers to a widget are currently implementation-dependent. Clients must therefore be aware that storing **False** into the *continue_to_dispatch* argument can lead to portability problems.

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*;
    XtPointer *client_data*;

w Specifies the widget for which this event handler is being registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies whether this procedure should be called on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be called.

client_data Specifies additional data to be passed to the 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. Each widget has a single registered event handler list, which will contain any procedure--client_data pair exactly once regardless of the manner in which it is registered. If the procedure is already registered with the same *client_data* value, the specified mask augments the existing mask. If the widget is realized, **XtAddEventHandler** calls **XSelectInput**, if necessary. The order in which this procedure is called relative to other handlers registered for the same event is not defined.

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*;
    XtPointer *client_data*;

w Specifies the widget for which this procedure is registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to unregister this procedure.

nonmaskable Specifies whether this procedure should be removed on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be removed.

client_data Specifies the registered client data.

The **XtRemoveEventHandler** function unregisters an event handler registered with **XtAddEventHandler** or **XtInsertEventHandler** for the specified events. The request is ignored if *client_data* does not match the value given when the handler was registered. If the widget is realized and no other event handler requires the event, **XtRemoveEventHandler** calls **XSelectInput**. 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 registered with **XtAddEventHandler** or **XtInsertEventHandler** from receiving all selected events, call **XtRemoveEventHandler** with an *event_mask* of **XtAllEvents** and *nonmaskable* **True**. The procedure will continue to receive any events that have been specified in calls to **XtAddRawEventHandler** or **XtInsertRawEventHandler**.

To register an event handler procedure that receives events before or after all previously registered event handlers, use **XtInsertEventHandler**.

typedef enum {XtListHead, XtListTail} XtListPosition;

void XtInsertEventHandler(*w*, *event_mask*, *nonmaskable*, *proc*, *client_data*, *position*)
    Widget *w*;
    EventMask *event_mask*;
    Boolean *nonmaskable*;
    XtEventHandler *proc*;
    XtPointer *client_data*;
    XtListPosition *position*;

w Specifies the widget for which this event handler is being registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies whether this procedure should be called on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be called.

client_data Specifies additional data to be passed to the client's event handler.

position Specifies when the event handler is to be called relative to other previously registered handlers.

XtInsertEventHandler is identical to **XtAddEventHandler** with the additional *position* argument. If *position* is **XtListHead**, the event handler is registered so that it will be called before any event handlers that were previously registered for the same widget. If *position* is **XtListTail**, the event handler is registered to be called after any previously registered event handlers. If the procedure is already registered with the same *client_data* value, the specified mask augments the existing mask and the procedure is repositioned in the list.

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 explicitly causing the X 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*;
    XtPointer *client_data*;

w Specifies the widget for which this event handler is being registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies whether this procedure should be called on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be called.

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 event 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. If the procedure is already registered with the same *client_data*, the specified mask augments the existing mask. The order in which this procedure is called relative to other handlers registered for the same event is not defined.

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*;
    XtPointer *client_data*;

w Specifies the widget for which this procedure is registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to unregister this procedure.

nonmaskable Specifies whether this procedure should be removed on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be registered.

client_data Specifies the registered client data.

The **XtRemoveRawEventHandler** function unregisters an event handler registered with **XtAddRawEventHandler** or **XtInsertRawEventHandler** for the specified events without changing the window event mask. The request is ignored if *client_data* does not match the value given when the handler was registered. If the specified procedure has not been registered or if it has been registered with a different value of *client_data*, **XtRemoveRawEventHandler** returns without reporting an error.

To stop a procedure registered with **XtAddRawEventHandler** or **XtInsertRawEventHandler** from receiving all nonselected events, call **XtRemoveRawEventHandler** with an *event_mask* of **XtAllEvents** and *nonmaskable* **True**. The procedure will continue to receive any events that have been specified in calls to **XtAddEventHandler** or **XtInsertEventHandler**.

To register an event handler procedure that receives events before or after all previously registered event handlers without selecting for the events, use **XtInsertRawEventHandler**.

void XtInsertRawEventHandler(*w*, *event_mask*, *nonmaskable*, *proc*, *client_data*, *position*)
  Widget *w*;
  EventMask *event_mask*;
  Boolean *nonmaskable*;
  XtEventHandler *proc*;
  XtPointer *client_data*;
  XtListPosition *position*;

w Specifies the widget for which this event handler is being registered. Must be of class Core or any subclass thereof.

event_mask Specifies the event mask for which to call this procedure.

nonmaskable Specifies whether this procedure should be called on the nonmaskable events (**GraphicsExpose**, **NoExpose**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, **ClientMessage**, and **MappingNotify**).

proc Specifies the procedure to be registered.

client_data Specifies additional data to be passed to the client's event handler.

position Specifies when the event handler is to be called relative to other previously registered handlers.

The **XtInsertRawEventHandler** function is similar to **XtInsertEventHandler** except that it does not modify the widget's event mask and never causes an **XSelectInput** for the specified events. If the procedure is already registered with the same *client_data* value, the specified mask augments the existing mask and the procedure is repositioned in the list.

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. Must be of class Core or any subclass thereof.

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 **XtInsertEventHandler** 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.

Chapter 8

Callbacks

Applications and other widgets often need to register a procedure with a widget that gets called under certain prespecified 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 an XtNdestroyCallbacks callback list resource. Widgets can define additional callback lists as they see fit. For example, the Pushbutton widget has a callback list to notify clients when the button has been activated.

Except where otherwise noted, it is the intent that all Intrinsics functions may be called at any time, including from within callback procedures, action routines, and event handlers.

8.1. Using Callback Procedure and Callback List Definitions

Callback procedure pointers for use in callback lists are of type **XtCallbackProc**.

```
typedef void (*XtCallbackProc)(Widget, XtPointer, XtPointer);
      Widget w;
      XtPointer client_data;
      XtPointer call_data;
```

w	Specifies the widget owning the list in which the callback is registered.
client_data	Specifies additional data supplied by the client when the procedure was registered.
call_data	Specifies any callback-specific data the widget wants to pass to the client. For example, when Scrollbar executes its XtNthumbChanged callback list, it passes the new position of the thumb.

The *client_data* argument provides a way for the client registering the callback procedure 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 may 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 always 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;
        XtPointer 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, (XtPointer) clientDataA},
        {B, (XtPointer) clientDataB},
        {(XtCallbackProc) NULL, (XtPointer) NULL}
};
```

Although callback lists are passed by address in arglists and varargs lists, the Intrinsics recognize callback lists throught the widget resource list and will copy the contents when necessary. Widget initialize and set_values procedures should not allocate memory for the callback list contents. The Intrinsics automatically do this, potentially 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 except **XtCallCallbackList** 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 must be declared with a resource type of **XtRCallback**. The internal representation of a callback list is implementation-dependent; widgets may make no assumptions about the value stored in this resource if it is non-NULL. Except to compare the value to NULL (which is equivalent to **XtCallbackStatus XtCallbackHasNone**), access to callback list resources must be made through other Intrinsics procedures.

8.3. Adding Callback Procedures

To add a callback procedure to a widget's callback list, use **XtAddCallback**.

void XtAddCallback(*w*, *callback_name*, *callback*, *client_data*)
    Widget *w*;
    String *callback_name*;
    XtCallbackProc *callback*;
    XtPointer *client_data*;

w	Specifies the widget. Must be of class Object or any subclass thereof.
callback_name	Specifies the callback list to which the procedure is to be appended.
callback	Specifies the callback procedure.
client_data	Specifies additional data to be passed to the specified procedure when it is invoked, 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. Must be of class Object or any subclass thereof.
callback_name	Specifies the callback list to which the procedures are 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 widget's callback list, use **XtRemoveCallback**.

void XtRemoveCallback(*w*, *callback_name*, *callback*, *client_data*)
  Widget *w*;
  String *callback_name*;
  XtCallbackProc *callback*;
  XtPointer *client_data*;

w Specifies the widget. Must be of class Object or any subclass thereof.

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 with the registered callback entry.

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. Must be of class Object or any subclass thereof.

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. Must be of class Object or any subclass thereof.

callback_name Specifies the callback list to be cleared.

8.5. Executing Callback Procedures

To execute the procedures in a given widget's callback list, specifying the callback list by resource name, use **XtCallCallbacks**.

void XtCallCallbacks(*w*, *callback_name*, *call_data*)
  Widget *w*;
  String *callback_name*;
  XtPointer *call_data*;

w Specifies the widget. Must be of class Object or any subclass thereof.

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, or NULL.

XtCallCallbacks calls each of the callback procedures in the list named by *callback_name* in the specified widget, passing the client data registered with the procedure and *call-data*.

To execute the procedures in a callback list, specifying the callback list by address, use **XtCallCallbackList**.

void XtCallCallbackList(*widget*, *callbacks*, *call_data*)
  Widget *widget*;
  XtCallbackList *callbacks*;
  XtPointer *call_data*;

widget Specifies the widget instance that contains the callback list. Must be of class Object or any subclass thereof.

callbacks Specifies the callback list to be executed.

call_data Specifies a callback-list-specific data value to pass to each of the callback procedures in the list, or NULL.

The *callbacks* parameter must specify the contents of a widget or object resource declared with representation type **XtRCallback**. If *callbacks* is NULL, **XtCallCallbackList** returns immediately; otherwiese it calls each of the callback procedures in the list, passing the client data and *call_data*.

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. Must be of class Object or any subclass thereof.

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**.

Chapter 9

Resource Management

A resource is a field in the widget record with a corresponding resource entry in the *resources* 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 a resource file (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).

Widgets typically 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 by 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 last, 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;
	XtPointer default_addr;
} XtResource, *XtResourceList;
```

When the resource list is specified as the **CoreClassPart**, **ObjectClassPart**, **RectObjClassPart**, or **ConstraintClassPart** *resources* field the strings pointed to by *resource_name*, *resource_Iclass*, *resource_type*, and *default_type* must be permanently allocated prior to or during the execution of the class initialization procedure and must not be subsequently deallocated.

The *resource_name* field contains the name used by clients to access the field in the widget. By convention, it starts with a lower-case letter and is spelled exactly like the field name, except all underscores (_) are deleted and the next letter is replaced by its upper-case counterpart. For example, the resource name for background_pixel becomes backgroundPixel. Resource names beginning with the two-character sequence ‘‘xt’’ and resource classes beginning with the two-character sequence ‘‘Xt’’ are reserved to the Intrinsics for future standard and implementation-dependent uses. 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 <**X11/StringDefs.h**>. The Intrinsics’s symbolic resource names begin with ‘‘XtN’’ and are followed by the string name (for example, XtNbackgroundPixel for backgroundPixel).

The *resource_class* field contains the class string used in resource specification files to identify the field. 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 the 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 font resources (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 to distinguish them from resource names. Their symbolic names are preceded with ‘‘XtC’’ (for example, XtCBackground).

The *resource_type* field gives the physical representation type of the resource and also encodes information about the specific usage of the field. By convention, it starts with an upper-case 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 format of the resource default value (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
XtRAtom	Atom
XtRBitmap	Pixmap, depth=1
XtRBoolean	Boolean
XtRBool	Bool
XtRCallback	XtCallbackList
XtRCardinal	Cardinal
XtRColor	XColor
XtRColormap	Colormap
XtRCursor	Cursor
XtRDimension	Dimension
XtRDisplay	Display*
XtREnum	XtEnum
XtRFile	FILE*
XtRFloat	float
XtRFont	Font
XtRFontSet	XFontSet
XtRFontStruct	XFontStruct*

Resource Type	Structure or Field Type
XtRFunction	(*)()
XtRGeometry	char*, format as defined by **XParseGeometry**
XtRInitialState	int
XtRInt	int
XtRLongBoolean	long
XtRObject	Object
XtRPixel	Pixel
XtRPixmap	Pixmap
XtRPointer	XtPointer
XtRPosition	Position
XtRScreen	Screen*
XtRShort	short
XtRString	String
XtRStringArray	String*
XtRStringTable	String*
XtRTranslationTable	XtTranslations
XtRUnsignedChar	unsigned char
XtRVisual	Visual*
XtRWidget	Widget
XtRWidgetClass	WidgetClass
XtRWidgetList	WidgetList
XtRWindow	Window

<**X11/StringDefs.h**> also defines the following resource types as a convenience for widgets, although they do not have any corresponding data type assigned: **XtREditMode**, **XtRJustify**, and **XtROrientation**.

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 **XtOffsetOf** 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 value 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 for which the converter is guaranteed to work (for example, **XtDefaultForeground** for a pixel resource). The *default_addr* field specifies the address of the default resource value. As a special case, if *default_type* is **XtRString**, then the value in the *default_addr* field is the pointer to the string rather than a pointer to the pointer. 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_addr* field is the actual value of the resource rather than the address of the value. The value must be in the correct representation type for the resource, coerced to an **XtPointer**. No conversion is possible, since there is no source representation type. XtRCallProc indicates that the value in the *default_addr* field is a procedure pointer. This procedure is automatically invoked with the widget, *resource_offset*, and a pointer to an **XrmValue** in which to store the result. XtRCallProc procedure pointers are of type **XtResourceDefaultProc**.

typedef void (*XtResourceDefaultProc)(Widget, int, XrmValue*);
  Widget *w*;
  int *offset*;
  XrmValue **value*;

w Specifies the widget whose resource value is to be obtained.

offset Specifies the offset of the field in the widget record.

value Specifies the resource value descriptor to return.

The **XtResourceDefaultProc** procedure should fill in the *value->addr* field with a pointer to the resource value in its correct representation type.

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*;

class Specifies the object class to be queried. It must be **objectClass** or any subclass thereof.

resources_return Returns the resource list.

num_resources_return Returns the number of entries in the resource list.

If **XtGetResourceList** is called before the class is initialized, it returns the resource list as specified in the class record. If it is called after the class has been initialized, **XtGetResourceList** returns a merged resource list that includes the resources for all superclasses. The list returned by **XtGetResourceList** should be freed using **XtFree** when it is no longer needed.

To get the constraint resource list structure for a particular widget class, use **XtGetConstraintResourceList**.

void XtGetConstraintResourceList(*class*, *resources_return*, *num_resources_return*)
  WidgetClass *class*;
  XtResourceList **resources_return*;
  Cardinal **num_resources_return*;

class Specifies the object class to be queried. It must be **objectClass** or any subclass thereof.

resources_return Returns the constraint resource list.

num_resources_return Returns the number of entries in the constraint resource list.

If **XtGetConstraintResourceList** is called before the widget class is initialized, the resource list as specified in the widget class Constraint part is returned. If **XtGetConstraintResourceList** is called after the widget class has been initialized, the merged resource list for the class and all Constraint superclasses is returned. If the specified class is not a subclass of **constraintWidgetClass**, **resources_return* is set to NULL and **num_resources_return* is set to zero. The list returned by **XtGetConstraintResourceList** should be freed using **XtFree** when it is no longer needed.

The routines **XtSetValues** and **XtGetValues** also use the resource list to set and get widget state; see Sections 9.7.1 and 9.7.2.

Here is an abbreviated version of a possible resource list for a Label widget:

```
/* Resources specific to Label */
static XtResource resources[] = {
```

```
{XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
  XtOffsetOf(LabelRec, label.foreground), XtRString, XtDefaultForeground},
{XtNfont, XtCFont, XtRFontStruct, sizeof(XFontStruct*),
  XtOffsetOf(LabelRec, label.font), XtRString, XtDefaultFont},
{XtNlabel, XtCLabel, XtRString, sizeof(String),
  XtOffsetOf(LabelRec, 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 top of the widget tree, the instance name of the widget itself, and the resource name of the specified field of the widget. Similarly, 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 top of the widget tree, the widget class name of the widget itself, and the resource class of the specified field of the widget.

9.2. Byte Offset Calculations

To determine the byte offset of a field within a structure type, use **XtOffsetOf**.

Cardinal XtOffsetOf(*structure_type*, *field_name*)
    Type structure_type;
    Field field_name;

structure_type Specifies a type that is declared as a structure.

field_name Specifies the name of a member within the structure.

The **XtOffsetOf** macro expands to a constant expression that gives the offset in bytes to the specified structure member from the beginning of the structure. It is normally used to statically initialize resource lists and is more portable than **XtOffset**, which serves the same function.

To determine the byte offset of a field within a structure pointer type, 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 a structure.

field_name Specifies the name of a member within the structure.

The **XtOffset** macro expands to a constant expression that gives the offset in bytes to the specified structure member from the beginning of the structure. It may be used to statically initialize resource lists. **XtOffset** is less portable than **XtOffsetOf**.

9.3. Superclass-to-Subclass Chaining of Resource Lists

The **XtCreateWidget** function gets resources as a superclass-to-subclass chained operation. That is, the resources specified in the **objectClass** resource list are fetched, then those in **rectObjClass**, 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. If reordering is necessary, a copy of the superclass resource list is made to avoid affecting other subclasses of the superclass.

Also at class initialization time, the Intrinsics produce an internal representation of the resource list to optimize access time when creating widgets. In order to save memory, the Intrinsics may overwrite the storage allocated for the resource list in the class record; therefore, widgets must allocate resource lists in writable storage and must not access the list contents directly after the class_initialize procedure has returned.

9.4. Subresources

A widget does not do anything to retrieve 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. Such widgets call **XtGetSubresources** to accomplish this.

void XtGetSubresources(*w*, *base*, *name*, *class*, *resources*, *num_resources*, *args*, *num_args*)
    Widget *w*;
    XtPointer *base*;
    String *name*;
    String *class*;
    XtResourceList *resources*;
    Cardinal *num_resources*;
    ArgList *args*;
    Cardinal *num_args*;

w	Specifies the object used to qualify the subpart resource name and class. Must be of class Object or any subclass thereof.
base	Specifies the base address of the subpart data structure into which the resources will 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 entries in the resource list.
args	Specifies the argument list to override any other resource specifications.
num_args	Specifies the number of entries in the argument list.

The **XtGetSubresources** function constructs a name and class list from the application name and class, the names and classes of all the object's ancestors, and the object itself. Then it appends to this list the *name* and *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.

XtGetSubresources may overwrite the specified resource list with an equivalent representation in an internal format, which optimizes access time if the list is used repeatedly. The resource list must be allocated in writable storage, and the caller must not modify the list contents after the call if the same list is to be used again. Resources fetched by **XtGetSubresources** are reference-counted as if they were referenced by the specified object. Subresources might therefore be freed from the conversion cache and destroyed when the object is destroyed, but not before then.

To fetch resources for widget subparts using varargs lists, use **XtVaGetSubresources**.

void XtVaGetSubresources(*w*, *base*, *name*, *class*, *resources*, *num_resources*, ...)
  Widget *w*;
  XtPointer *base*;
  String *name*;
  String *class*;
  XtResourceList *resources*;
  Cardinal *num_resources*;

w	Specifies the object used to qualify the subpart resource name and class. Must be of class Object or any subclass thereof.
base	Specifies the base address of the subpart data structure into which the resources will 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 entries in the resource list.
...	Specifies the variable argument list to override any other resource specifications.

XtVaGetSubresources is identical in function to **XtGetSubresources** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

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*;
  XtPointer *base*;
  XtResourceList *resources*;
  Cardinal *num_resources*;
  ArgList *args*;
  Cardinal *num_args*;

w	Specifies the object that identifies the resource database to search (the database is that associated with the display for this object). Must be of class Object or any subclass thereof.
base	Specifies the base address into which the resource values will be written.
resources	Specifies the resource list.
num_resources	Specifies the number of entries in the resource list.
args	Specifies the argument list to override any other resource specifications.
num_args	Specifies the number of entries in the argument list.

The **XtGetApplicationResources** function first uses the passed object, which is usually an application shell widget, to construct a resource name and class list. The full name and class of the specified object (that is, including its ancestors, if any) is logically added to the front of each resource name and class. 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 addresses obtained by adding *base* to each *offset* 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.

XtGetApplicationResources may overwrite the specified resource list with an equivalent representation in an internal format, which optimizes access time if the list is used repeatedly. The resource list must be allocated in writable storage, and the caller must not modify the list contents after the call if the same list is to be used again. Any per-display resources fetched by **XtGetApplicationResources** will not be freed from the resource cache until the display is closed.

To retrieve resources for the overall application using varargs lists, use **XtVaGetApplicationResources**.

void XtVaGetApplicationResources(*w*, *base*, *resources*, *num_resources*, ...)
    Widget *w*;
    XtPointer *base*;
    XtResourceList *resources*;
    Cardinal *num_resources*;

w	Specifies the object that identifies the resource database to search (the database is that associated with the display for this object). Must be of class Object or any subclass thereof.
base	Specifies the base address into which the resource values will be written.
resources	Specifies the resource list for the subpart.
num_resources	Specifies the number of entries in the resource list.
...	Specifies the variable argument list to override any other resource specifications.

XtVaGetApplicationResources is identical in function to **XtGetApplicationResources** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

9.6. Resource Conversions

The Intrinsics provide a mechanism for registering representation converters that are automatically invoked by the resource-fetching routines. The Intrinsics additionally provide and register several commonly used converters. This resource conversion mechanism serves several purposes:

- It permits user and application resource files to contain textual representations of nontextual values.
- It allows textual or other representations of default resource values that are dependent on the display, screen, or colormap, and thus must be computed at runtime.
- It caches 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 string-to-color) are performed only once.

9.6.1. Predefined Resource Converters

The Intrinsics define all the representations used in the Object, RectObj, Core, Composite, Constraint, and Shell widget classes. The Intrinsics register the following resource converters that accept input values of representation type **XtRString**.

Target Representation	Converter Name	Additional Args
XtRAcceleratorTable	**XtCvtStringToAcceleratorTable**	

XtRAtom	**XtCvtStringToAtom**	Display*
XtRBoolean	**XtCvtStringToBoolean**	
XtRBool	**XtCvtStringToBool**	
XtRCursor	**XtCvtStringToCursor**	Display*
XtRDimension	**XtCvtStringToDimension**	
XtRDisplay	**XtCvtStringToDisplay**	
XtRFile	**XtCvtStringToFile**	
XtRFloat	**XtCvtStringToFloat**	
XtRFont	**XtCvtStringToFont**	Display*
XtRFontSet	**XtCvtStringToFontSet**	Display*, String *locale*
XtRFontStruct	**XtCvtStringToFontStruct**	Display*
XtRInitialState	**XtCvtStringToInitialState**	
XtRInt	**XtCvtStringToInt**	
XtRPixel	**XtCvtStringToPixel**	**colorConvertArgs**
XtRPosition	**XtCvtStringToPosition**	
XtRShort	**XtCvtStringToShort**	
XtRTranslationTable	**XtCvtStringToTranslationTable**	
XtRUnsignedChar	**XtCvtStringToUnsignedChar**	
XtRVisual	**XtCvtStringToVisual**	Screen*, Cardinal *depth*

The String-to-Pixel conversion has two predefined constants that are guaranteed to work and contrast with each other: **XtDefaultForeground** and **XtDefaultBackground**. They evaluate to the black and white pixel values of the widget's screen, respectively. If the application resource reverseVideo is **True**, they evaluate to the white and black pixel values of the widget's screen, respectively. Similarly, the String-to-Font and String-to-FontStruct converters recognize the constant **XtDefaultFont** and evaluate this in the following manner:

- Query the resource database for the resource whose full name is ''xtDefaultFont'', class ''XtDefaultFont'' (that is, no widget name/class prefixes) and use a type **XtRString** value returned as the font name, or a type **XtRFont** or **XtRFontStruct** value directly as the resource value.
- If the resource database does not contain a value for xtDefaultFont, class XtDefaultFont, or if the returned font name cannot be successfully opened, an implementation-defined font in ISO8859-1 character set encoding is opened. (One possible algorithm is to perform an **XListFonts** using a wildcard font name and use the first font in the list. This wildcard font name should be as broad as possible to maximize the probability of locating a useable font; for example, "-*-*-*-R-*-*-*-120-*-*-*-*-ISO8859-1".)
- If no suitable ISO8859-1 font can be found, issue a warning message and return **False**.

The String-to-FontSet converter recognizes the constant **XtDefaultFontSet** and evaluate this in the following manner:

- Query the resource database for the resource whose full name is ''xtDefaultFontSet'', class ''XtDefaultFontSet'' (that is, no widget name/class prefixes) and use a type **XtRString** value returned as the base font name list, or a type **XtRFontSet** value directly as the resource value.
- If the resource database does not contain a value for xtDefaultFontSet, class XtDefault-FontSet, or if a font set cannot be successfully created from this resource, an implementation-defined font set is created. (One possible algorithm is to perform an **XCreateFontSet** using a wildcard base font name. This wildcard base font name should be as broad as possible to maximize the probability of locating a useable font; for example, "-*-*-*-R-*-*-*-120-*-*-*-*".)
- If no suitable font set can be created, issue a warning message and return **False**.

If a font set is created but *missing_charset_list* is not empty, a warning is issued and the partial font set is returned. The Intrinsics register the String-to-FontSet converter with a conversion argument list that extracts the current process locale at the time the converter is invoked. This ensures that the converter is invoked again if the same conversion is required in a different locale.

The String-to-InitialState conversion accepts the values **NormalState** or **IconicState** as defined by the *Inter-Client Communication Conventions Manual*.

The String-to-Visual conversion calls **XMatchVisualInfo** using the *screen* and *depth* fields from the core part and returns the first matching Visual on the list. The widget resource list must be certain to specify any resource of type **XtRVisual** after the depth resource. The allowed string values are the visual class names defined in *X Window System Protocol*, Section 8; **StaticGray**, **StaticColor**, **TrueColor**, **GrayScale**, **PseudoColor**, and **DirectColor**.

The Intrinsics register the following resource converter that accepts an input value of representation type **XtRColor**.

Target Representation	Converter Name	Additional Args
XtRPixel	**XtCvtColorToPixel**	

The Intrinsics register the following resource converters that accept input values of representation type **XtRInt**.

Target Representation	Converter Name	Additional Args
XtRBoolean	**XtCvtIntToBoolean**	
XtRBool	**XtCvtIntToBool**	
XtRColor	**XtCvtIntToColor**	**colorConvertArgs**
XtRDimension	**XtCvtIntToDimension**	
XtRFloat	**XtCvtIntToFloat**	
XtRFont	**XtCvtIntToFont**	
XtRPixel	**XtCvtIntToPixel**	
XtRPixmap	**XtCvtIntToPixmap**	
XtRPosition	**XtCvtIntToPosition**	
XtRShort	**XtCvtIntToShort**	
XtRUnsignedChar	**XtCvtIntToUnsignedChar**	

The Intrinsics register the following resource converter that accepts an input value of representation type **XtRPixel**.

Target Representation	Converter Name	Additional Args
XtRColor	**XtCvtPixelToColor**	

9.6.2. New Resource Converters

Type converters use pointers to **XrmValue** structures (defined in <**X11/Xresource.h**>; see Section 15.4 in *Xlib – C Language X Interface*) for input and output values.

```
typedef struct {
        unsigned int size;
        XPointer addr;
} XrmValue, *XrmValuePtr;
```

The *addr* field specifies the address of the data and the *size* field gives the total number of significant bytes in the data. For values of type **String**, *addr* is the address of the first character and *size* includes the NUL terminating byte.

A resource converter procedure pointer is of type **XtTypeConverter**.

```
typedef Boolean (*XtTypeConverter)(Display*, XrmValue*, Cardinal*,
                        XrmValue*, XrmValue*, XtPointer*);
    Display *display;
    XrmValue *args;
    Cardinal *num_args;
    XrmValue *from;
    XrmValue *to;
    XtPointer *converter_data;
```

display Specifies the display connection with which this conversion is associated.

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*, and the String-to-Pixel converter needs the widget's *screen* and *colormap*.

num_args Specifies the number of entries in *args*.

from Specifies the value to convert.

to Specifies a descriptor for a location into which to store the converted value.

converter_data Specifies a location into which the converter may store converter-specific data associated with this conversion.

The *display* argument is normally used only when generating error messages, to identify the application context (with the function **XtDisplayToApplicationContext**).

The *to* argument specifies the size and location into which the converter should store the converted value. If the *addr* field is NULL, the converter should allocate appropriate storage and store the size and location into the *to* descriptor. If the type converter allocates the storage, it remains under the ownership of the converter and must not be modified by the caller. The type converter is permitted to use static storage for this purpose, and therefore the caller must immediately copy the data upon return from the converter. If the *addr* field is not NULL, the converter must check the *size* field to ensure that sufficient space has been allocated before storing the converted value. If insufficient space is specified, the converter should update the *size* field with the number of bytes required and return **False** without modifying the data at the specified location. If sufficient space was allocated by the caller, the converter should update the *size* field with the number of bytes actually occupied by the converted value. For converted values of type **XtRString**, the size should include the NULL terminating byte, if any. The converter may store any value in the location specified in *converter_data*; this data will be passed to the destructor, if any, when the resource is freed by the Intrinsics.

The converter must return **True** if the conversion was successful and **False** otherwise. If the conversion cannot be performed because of an improper source value, a warning message should also be issued with **XtAppWarningMsg**.

Most type converters just take the data described by the specified *from* argument and return data by writing into the location specified in the *to* argument. A few need other information, which is available in *args*. 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 if an address is written into *to->addr*, it cannot be that of a local variable of the converter because the data will not be valid after the converter returns. Static variables may be used, as in the following example. If the converter modifies the resource database, the changes affect any in-progress widget creation, **XtGetApplicationResources**, or **XtGetSubresources** in an implementation-defined manner; however, insertion of new entries or changes to existing entries is allowed and will not directly cause an error.

The following is an example of a converter that takes a **string** and converts it to a **Pixel**. Note that the *display* parameter is only used to generate error messages; the **Screen** conversion argument is still required to inform the Intrinsics that the converted value is a function of the particular display (and colormap).

```
#define done(type, value) \
	{							\
		if (toVal->addr != NULL) {			\
			if (toVal->size < sizeof(type)) {	\
				toVal->size = sizeof(type);	\
				return False;			\
			}					\
			*(type*)(toVal->addr) = (value);	\
		}						\
		else {						\
			static type static_val;			\
			static_val = (value);			\
			toVal->addr = (XPointer)&static_val;	\
		}						\
		toVal->size = sizeof(type);			\
		return True;					\
	}

static Boolean CvtStringToPixel(dpy, args, num_args, fromVal, toVal, converter_data)
	Display	*dpy;
	XrmValue *args;
	Cardinal	*num_args;
	XrmValue *fromVal;
	XrmValue *toVal;
	XtPointer	*converter_data;
{
	static XColor	screenColor;
	XColor		exactColor;
	Screen		*screen;
	Colormap	colormap;
	Status		status;
	char		message[1000];

	if (*num_args != 2)
		XtAppErrorMsg(XtDisplayToApplicationContext(dpy),
			"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);

    if (strcmp(message, "xtdefaultbackground") == 0) done(&WhitePixelOfScreen(screen), Pixel);
    if (strcmp(message, "xtdefaultforeground") == 0) done(&BlackPixelOfScreen(screen), Pixel);

    status = XAllocNamedColor(DisplayOfScreen(screen), colormap, (char*)fromVal->addr,
                    &screenColor, &exactColor);

    if (status == 0) {
        String params[1];
        Cardinal num_params = 1;
        params[0] = (String)fromVal->addr;
        XtAppWarningMsg(XtDisplayToApplicationContext(dpy),
            "cvtStringToPixel", "noColormap", "XtToolkitError",
            "Cannot allocate colormap entry for \"%s\"", params, &num_params);
    } else {
        done( &screenColor.pixel, Pixel );
    }

    /* converter_data not used here */
};
```

All type converters should define some set of conversion values for which they are guaranteed to succeed 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 in the same manner as **XtDefaultForeground**, **XtDefaultBackground**, and **XtDefaultFont**.

To allow the Intrinsics to deallocate resources produced by type converters, a resource destructor procedure may also be provided.

A resource destructor procedure pointer is of type **XtDestructor**.

typedef void (*XtDestructor) (XtAppContext, XrmValue*, XtPointer, XrmValue*, Cardinal*);
  XtAppContext *app*;
  XrmValue ***to*;
  XtPointer *converter_data*;
  XrmValue **args*;
  Cardinal **num_args*;

app	Specifies an application context in which the resource is being freed.
to	Specifies a descriptor for the resource produced by the type converter.
converter_data	Specifies the converter-specific data returned by the type converter.
args	Specifies the additional converter arguments as passed to the type converter when the conversion was performed.
num_args	Specifies the number of entries in *args*.

The destructor procedure is responsible for freeing the resource specified by the *to* argument, including any auxiliary storage associated with that resource, but not the memory directly addressed by the size and location in the *to* argument nor the memory specified by *args*.

9.6.3. Issuing Conversion Warnings

The **XtDisplayStringConversionWarning** procedure is a convenience routine for resource type converters that convert from string values.

void XtDisplayStringConversionWarning(*display*, *from_value*, *to_type*)
    Display **display*;
    String *from_value*, *to_type*;

display Specifies the display connection with which the conversion is associated.

from_value Specifies the string that could not be converted.

to_type Specifies the target representation type requested.

The **XtDisplayStringConversionWarning** procedure issues a warning message using **XtAppWarningMsg** with *name* ‘‘conversionError’’, *type* ‘‘string’’, *class* ‘‘XtToolkitError’’, and the default message ‘‘Cannot convert "*from_value*" to type *to_type*’’.

To issue other types of warning or error messages, the type converter should use **XtAppWarningMsg** or **XtAppErrorMsg**.

To retrieve the application context associated with a given display connection, use **XtDisplayToApplicationContext**.

XtAppContext XtDisplayToApplicationContext(*display*)
    Display **display*;

display Specifies an open and initialized display connection.

The **XtDisplayToApplicationContext** function returns the application context in which the specified *display* was initialized. If the display is not known to the Intrinsics, an error message is issued.

9.6.4. Registering a New Resource Converter

When registering a resource converter, the client must specify the manner in which the conversion cache is to be used when there are multiple calls to the converter. Conversion cache control is specified via an **XtCacheType** argument.

typedef int XtCacheType;

An **XtCacheType** field may contain one of the following values:

XtCacheNone

Specifies that the results of a previous conversion may not be reused to satisfy any other resource requests; the specified converter will be called each time the converted value is required.

XtCacheAll

Specifies that the results of a previous conversion should be reused for any resource request that depends upon the same source value and conversion arguments.

XtCacheByDisplay

Specifies that the results of a previous conversion should be used as for **XtCacheAll** but the destructor will be called, if specified, if **XtCloseDisplay** is called for the display connection associated with the converted value, and the value will be removed from the conversion cache.

The qualifier **XtCacheRefCount** may be ORed with any of the above values. If **XtCacheRefCount** is specified, calls to **XtCreateWidget**, **XtCreateManagedWidget**, **XtGetApplicationResources** and **XtGetSubresources** that use the converted value will be counted. When a widget using the converted value is destroyed, the count is decremented, and if the count reaches zero, the destructor procedure will be called and the converted value will be removed from the conversion cache.

To register a type converter for all application contexts in a process, use **XtSetTypeConverter** and to register a type converter in a single application context, use **XtAppSetTypeConverter**.

void XtSetTypeConverter(*from_type*, *to_type*, *converter*, *convert_args*, *num_args*,
    cache_type, *destructor*)
  String *from_type*;
  String *to_type*;
  XtTypeConverter *converter*;
  XtConvertArgList *convert_args*;
  Cardinal *num_args*;
  XtCacheType *cache_type*;
  XtDestructor *destructor*;

from_type Specifies the source type.

to_type Specifies the destination type.

converter Specifies the resource type converter procedure.

convert_args Specifies additional conversion arguments, or NULL.

num_args Specifies the number of entries in *convert_args*.

cache_type Specifies whether or not resources produced by this converter are sharable or display-specific and when they should be freed.

destructor Specifies a destroy procedure for resources produced by this conversion, or NULL if no additional action is required to deallocate resources produced by the converter.

void XtAppSetTypeConverter(*app_context*, *from_type*, *to_type*, *converter*, *convert_args*,
    num_args, *cache_type*, *destructor*)
  XtAppContext *app_context*;
  String *from_type*;
  String *to_type*;
  XtTypeConverter *converter*;
  XtConvertArgList *convert_args*;
  Cardinal *num_args*;
  XtCacheType *cache_type*;
  XtDestructor *destructor*;

app_context Specifies the application context.

from_type Specifies the source type.

to_type Specifies the destination type.

converter Specifies the resource type converter procedure.

convert_args Specifies additional conversion arguments, or NULL.

num_args Specifies the number of entries in *convert_args*.

cache_type Specifies whether or not resources produced by this converter are sharable or display-specific and when they should be freed.

destructor Specifies a destroy procedure for resources produced by this conversion, or NULL if no additional action is required to deallocate resources produced by the converter.

XtSetTypeConverter registers the specified type converter and destructor in all application contexts created by the calling process, including any future application contexts that may be created. **XtAppSetTypeConverter** registers the specified type converter in the single application context specified. If the same *from_type* and *to_type* are specified in multiple calls to either function, the most recent overrides the previous ones.

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/Intrinsic.h**>.

```
typedef enum {
        /* address mode              parameter representation */
          XtAddress,                 /* address */
          XtBaseOffset,              /* offset */
          XtImmediate,               /* constant */
          XtResourceString,          /* resource name string */
          XtResourceQuark,           /* resource name quark */
          XtWidgetBaseOffset,        /* offset */
          XtProcedureArg             /* procedure to call */
} XtAddressMode;

typedef struct {
        XtAddressMode address_mode;
        XtPointer address_id;
        Cardinal size;
} XtConvertArgRec, *XtConvertArgList;
```

The *size* field specifies the length of the data in bytes. 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 the widget base. **XtResourceQuark** causes *address_id* to be interpreted as the result of an **XrmStringToQuark** conversion on the name of a resource, which is to be converted into an offset from the widget base. **XtWidgetBaseOffset** is similar to **XtBaseOffset** except that it searches for the closest windowed ancestor if the object is not of a subclass of Core (See Chapter 12). **XtProcedureArg** specifies that *address_id* is a pointer to a procedure to be invoked to return the conversion argument. If **XtProcedureArg** is specified, *address_id* must contain the address of a function of type **XtConvertArgProc**.

```
typedef void (*XtConvertArgProc)(Widget, Cardinal*, XrmValue*);
    Widget object;
    Cardinal *size;
    XrmValue *value;
```

object Passes the object for which the resource is being converted, or NULL if the converter was invoked by **XtCallConverter** or **XtDirectConvert**.

size Passes a pointer to the *size* field from the **XtConvertArgRec**.

value Passes a pointer to a descriptor into which the procedure must store the conversion argument.

When invoked, the **XtConvertArgProc** procedure must derive a conversion argument and store the address and size of the argument in the location pointed to by *value*.

In order to permit reentrancy, the **XtConvertArgProc** should return the address of storage whose lifetime is no shorter than the lifetime of *object*. If *object* is NULL, the lifetime of the conversion argument must be no shorter than the lifetime of the resource with which the conversion argument is associated. The Intrinsics do not guarantee to copy this storage but do guarantee not to reference it if the resource is removed from the conversion cache.

The following example illustrates how to register the CvtStringToPixel routine given earlier:

```
static XtConvertArgRec colorConvertArgs[] = {
    {XtWidgetBaseOffset, (XtPointer)XtOffset(Widget, core.screen),  sizeof(Screen*)},
    {XtWidgetBaseOffset, (XtPointer)XtOffset(Widget, core.colormap),sizeof(Colormap)}
};

XtSetTypeConverter(XtRString, XtRPixel, CvtStringToPixel,
   colorConvertArgs, XtNumber(colorConvertArgs), XtCacheByDisplay, NULL);
```

The conversion argument descriptors **colorConvertArgs** and **screenConvertArg** are predefined by the Intrinsics. Both take the values from the closest windowed ancestor if the object is not of a subclass of Core. 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 on which 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.

If any conversion arguments of type **XtBaseOffset**, **XtResourceString**, **XtResourceQuark**, and **XtWidgetBaseOffset** are specified for conversions performed by **XtGetApplicationResources**, **XtGetSubresources**, **XtVaGetApplicationResources** or **XtVaGetSubresources**, the arguments are computed with respect to the specified widget, not the base address or resource list specified in the call.

If the **XtConvertArgProc** modifies the resource database, the changes affect any in-progress widget creation, **XtGetApplicationResources**, or **XtGetSubresources** in an implementation-defined manner; however, insertion of new entries or changes to existing entries is allowed and will not directly cause an error.

9.6.5. Resource Converter Invocation

All resource-fetching routines (for example, **XtGetSubresources**, **XtGetApplicationResources**, and so on) call resource converters if the resource database or varargs list specifies a value that has a different representation from the desired representation or if the widget's default resource value representation is different from the desired representation.

To invoke explicit resource conversions, use **XtConvertAndStore** or **XtCallConverter**.

```
typedef XtPointer XtCacheRef;
```

Boolean XtCallConverter(*display*, *converter*, *conversion_args*, *num_args*, *from*, *to_in_out*, *cache_ref_return*)
    Display* *display*;
    XtTypeConverter *converter*;
    XrmValuePtr *conversion_args*;
    Cardinal *num_args*;
    XrmValuePtr *from*;
    XrmValuePtr *to_in_out*;
    XtCacheRef **cache_ref_return*;

display Specifies the display with which the conversion is to be associated.

converter Specifies the conversion procedure to be called.

conversion_args Specifies the additional conversion arguments needed to perform the conversion, or NULL.

num_args Specifies the number of entries in *conversion_args*.

from Specifies a descriptor for the source value.

to_in_out Returns the converted value.

cache_ref_return Returns a conversion cache id.

The **XtCallConverter** function looks up the specified type converter in the application context associated with the display and, if the converter was not registered or was registered with cache type **XtCacheAll** or **XtCacheByDisplay** looks in the conversion cache to see if this conversion procedure has been called with the specified conversion arguments. If so, it checks the success status of the prior call, and if the conversion failed, **XtCallConverter** returns **False** immediately; otherwise it checks the size specified in the *to* argument and, if it is greater than or equal to the size stored in the cache, copies the information stored in the cache into the location specified by *to->addr*, stores the cache size into *to->size*, and returns **True**. If the size specified in the *to* argument is smaller than the size stored in the cache, **XtCallConverter** copies the cache size into *to->size* and returns **False**. If the converter was registered with cache type **XtCacheNone** or no value was found in the conversion cache, **XtCallConverter** calls the converter and, if it was not registered with cache type **XtCacheNone**, enters the result in the cache. **XtCallConverter** then returns what the converter returned.

The *cache_ref_return* field specifies storage allocated by the caller in which an opaque value will be stored. If the type converter has been registered with the **XtCacheRefCount** modifier and if the value returned in *cache_ref_return* is non-NULL, then the caller should store the *cache_ref_return* value in order to decrement the reference count when the converted value is no longer required. The *cache_ref_return* argument should be NULL if the caller is unwilling or unable to store the value.

To explicitly decrement the reference counts for resources obtained from **XtCallConverter**, use **XtAppReleaseCacheRefs**.

void XtAppReleaseCacheRefs(*app_context*, *refs*)
    XtAppContext *app_context*;
    XtCacheRef **refs*;

app_context Specifies the application context.

refs Specifies the list of cache references to be released.

XtAppReleaseCacheRefs decrements the reference count for the conversion entries identified by the *refs* argument. This argument is a pointer to a NULL-terminated list of **XtCacheRef** values. If any reference count reaches zero, the destructor, if any, will be called and the resource removed from the conversion cache.

As a convenience to clients needing to explicitly decrement reference counts via a callback function, the Intrinsics define two callback procedures, **XtCallbackReleaseCacheRef** and **XtCallbackReleaseCacheRefList**.

void XtCallbackReleaseCacheRef(*object*, *client_data*, *call_data*)
  Widget *object*;
  XtPointer *client_data*;
  XtPointer *call_data*;

object Specifies the object with which the resource is associated.

client_data Specifies the conversion cache entry to be released.

call_data Is ignored.

This callback procedure may be added to a callback list to release a previously returned **XtCacheRef** value. When adding the callback, the callback *client_data* argument must be specified as the value of the **XtCacheRef** data cast to type **XtPointer**.

void XtCallbackReleaseCacheRefList(*object*, *client_data*, *call_data*)
  Widget *object*;
  XtPointer *client_data*;
  XtPointer *call_data*;

object Specifies the object with which the resources are associated.

client_data Specifies the conversion cache entries to be released.

call_data Is ignored.

This callback procedure may be added to a callback list to release a list of previously returned **XtCacheRef** values. When adding the callback, the callback *client_data* argument must be specified as a pointer to a NULL-terminated list of **XtCacheRef** values.

To lookup and call a resource converter, copy the resulting value, and free a cached resource when a widget is destroyed, use **XtConvertAndStore**.

Boolean XtConvertAndStore(*object*, *from_type*, *from*, *to_type*, *to_in_out*)
  Widget *object*;
  String *from_type*;
  XrmValuePtr *from*;
  String *to_type*;
  XrmValuePtr *to_in_out*;

object Specifies the object to use for additional arguments, if any are needed, and the destroy callback list. Must be of class Object or any subclass thereof.

from_type Specifies the source type.

from Specifies the value to be converted.

to_type Specifies the destination type.

to_in_out Specifies a descriptor for storage into which the converted value will be returned.

The **XtConvertAndStore** function looks up the type converter registered to convert *from_type* to *to_type*, computes any additional arguments needed, and then calls **XtCallConverter** (or **XtDirectConvert** if an old-style converter was registered with **XtAddConverter** or **XtAppAddConverter**; see Appendix C) with the *from* and *to_in_out* arguments. The *to_in_out* argument specifies the size and location into which the converted value will be stored and is passed directly to the converter. If the location is specified as NULL, it will be replaced with a pointer to private storage and the size will be returned in the descriptor. The caller is expected to copy this private storage immediately and must not modify it in any way. If a non-NULL location is specified, the caller must allocate sufficient storage to hold the converted value and must also specify the size of that storage in the descriptor. The *size* field will be modified on return to indicate the actual size of the converted data. If the conversion succeeds, **XtConvertAndStore** returns **True**; otherwise, it returns **False**.

XtConvertAndStore adds **XtCallbackReleaseCacheRef** to the destroyCallback list of the specified object if the conversion returns an **XtCacheRef** value. The resulting resource should not be referenced after the object has been destroyed.

XtCreateWidget performs processing equivalent to **XtConvertAndStore** when initializing the object instance. Because there is extra memory overhead required to implement reference counting, clients may distinguish those objects that are never destroyed before the application exits from those that may be destroyed and whose resources should be deallocated.

To specify whether reference counting is to be enabled for the resources of a particular object when the object is created, the client can specify a value for the **Boolean** resource XtNinitialResourcesPersistent, class XtCInitialResourcesPersistent.

When **XtCreateWidget** is called, if this resource is not specified as **False** in either the arglist or the resource database, then the resources referenced by this object are not reference-counted, regardless of how the type converter may have been registered. The effective default value is **True**; thus clients that expect to destroy one or more objects and want resources deallocated must explicitly specify **False** for XtNinitialResourcesPersistent.

The resources are still freed and destructors called when **XtCloseDisplay** is called if the conversion was registered as **XtCacheByDisplay**.

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 values of resources associated with a widget instance, use **XtGetValues**.

void XtGetValues(*object*, *args*, *num_args*)
    Widget *object*;
    ArgList *args*;
    Cardinal *num_args*;

object Specifies the object whose resource values are to be returned. Must be of class Object or any subclass thereof.

args Specifies the argument list of name/address pairs that contain the resource names and the addresses into which the resource values are to be stored. The resource names are widget-dependent.

num_args Specifies the number of entries in the argument list.

The **XtGetValues** function starts with the resources specified for the Object class and proceeds down the subclass chain to the class of the object. The *value* field of a passed argument list must contain the address into which to copy the contents of the corresponding object instance field. If the field is a pointer type, the lifetime of the pointed-to data is defined by the object class. For the Intrinsics-defined resources, the following lifetimes apply

- Not valid following any operation that modifies the resource:
  - XtNchildren resource of composite widgets.
  - All resources of representation type XtRCallback.
- Remain valid at least until the widget is destroyed:

  – XtNaccelerators, XtNtranslations.

- Remain valid until the Display is closed:
  – XtNscreen.

It is the caller's responsibility to allocate and deallocate storage for the copied data according to the size of the resource representation type used within the object.

If the class of the object'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 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. If any get_values_hook procedures in the object's class or superclass records are non-NULL, they are called in superclass-to-subclass order after all the resource values have been fetched by **XtGetValues**. Finally, if the object's parent is a subclass of **constraintWidgetClass**, and if any of the parent's class or superclass records have declared **ConstraintClassExtension** records in the Constraint class part *extension* field with a record type of **NULLQUARK** and if the *get_values_hook* field in the extension record is non-NULL, **XtGetValues** calls the get_values_hook procedures in superclass-to-subclass order. This permits a Constraint parent to provide nonresource data via **XtGetValues**.

Get_values_hook procedures may modify the data stored at the location addressed by the *value* field, including (but not limited to) making a copy of data whose resource representation is a pointer. None of the Intrinsics-defined object classes copy data in this manner. Any operation that modifies the queried object resource may invalidate the pointed-to data.

To retrieve the current values of resources associated with a widget instance using varargs lists, use **XtVaGetValues**.

```
void XtVaGetValues(object, ...)
      Widget object;
```

object	Specifies the object whose resource values are to be returned. Must be of class Object or any subclass thereof.
...	Specifies the variable argument list for the resources to be returned.

XtVaGetValues is identical in function to **XtGetValues** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1. All value entries in the list must specify pointers to storage allocated by the caller to which the resource value will be copied. It is the caller's responsibility to ensure that sufficient storage is allocated. If **XtVaTypedArg** is specified, the *type* argument specifies the representation desired by the caller and *the* size argument specifies the number of bytes allocated to store the result of the conversion. If the size is insufficient, a warning message is issued and the list entry is skipped.

9.7.1.1. Widget Subpart Resource Data: the get_values_hook Procedure

Widgets that have subparts can return resource values from them through **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 subpart resource values are to be retrieved.

args Specifies the argument list that was passed to **XtGetValues** or the transformed varargs list passed to **XtVaGetValues**.

num_args Specifies the number of entries in the argument list.

The widget with subpart resources should call **XtGetSubvalues** in the get_values_hook procedure and pass in its subresource list and the *args* and *num_args* parameters.

9.7.1.2. Widget Subpart State

To retrieve the current values of subpart resource data associated with a widget instance, use **XtGetSubvalues**. For a discussion of subpart resources, see Section 9.4.

void XtGetSubvalues(*base*, *resources*, *num_resources*, *args*, *num_args*)
  XtPointer *base*;
  XtResourceList *resources*;
  Cardinal *num_resources*;
  ArgList *args*;
  Cardinal *num_args*;

base Specifies the base address of the subpart data structure for which the resources should be retrieved.

resources Specifies the subpart resource list.

num_resources Specifies the number of entries in the resource list.

args Specifies the argument list of name/address pairs that contain the resource names and the addresses into which the resource values are to be stored.

num_args Specifies the number of entries in the argument list.

The **XtGetSubvalues** function obtains resource values from the structure identified by *base*. The *value* field in each argument entry must 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 subpart. If the argument list contains a resource name that is not found in the resource list, the value at the corresponding address is not modified.

To retrieve the current values of subpart resources associated with a widget instance using varargs lists, use **XtVaGetSubvalues**.

void XtVaGetSubvalues(*base*, *resources*, *num_resources*, ...)
  XtPointer *base*;
  XtResourceList *resources*;
  Cardinal *num_resources*;

base Specifies the base address of the subpart data structure for which the resources should be retrieved.

resources Specifies the subpart resource list.

num_resources Specifies the number of entries in the resource list.

... Specifies a variable argument list of name/address pairs that contain the resource names and the addresses into which the resource values are to be stored.

XtVaGetSubvalues is identical in function to **XtGetSubvalues** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1. **XtVaTypedArg** is not supported for **XtVaGetSubvalues**. If **XtVaTypedArg** is specified in the list, a warning message is issued and the entry is then ignored.

9.7.2. Setting Widget State

To modify the current values of resources associated with a widget instance, use **XtSetValues**.

void XtSetValues(*object*, *args*, *num_args*)
    Widget *object*;
    ArgList *args*;
    Cardinal *num_args*;

object Specifies the object whose resources are to be modified. Must be of class Object or any subclass thereof.

args Specifies the argument list of name/value pairs that contain the resources to be modified and their new values.

num_args Specifies the number of entries in the argument list.

The **XtSetValues** function starts with the resources specified for the Object class fields and proceeds down the subclass chain to the object. At each stage, it replaces the *object* resource fields with any values specified in the argument list. **XtSetValues** then calls the set_values procedures for the object in superclass-to-subclass order. If the object has any non-NULL *set_values_hook* fields, these are called immediately after the corresponding set_values procedure. This procedure permits subclasses to set subpart data via **XtSetValues**.

If the class of the object's parent is a subclass of **constraintWidgetClass**, **XtSetValues** also updates the object'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 replaces the constraint resource fields with any values specified in the argument list. 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 records, so that they can make adjustments to the desired values based on full information about the widget. Any arguments specified that do not match a resource list entry are silently ignored.

If the object is of a subclass of RectObj, **XtSetValues** determines if a geometry request is needed by comparing the old object to the new object. If any geometry changes are required, **XtSetValues** restores the original geometry and makes the request on behalf of the widget.a If the geometry manager returns **XtGeometryYes**, **XtSetValues** calls the object's resize procedure. If the geometry manager returns **XtGeometryDone**, **XtSetValues** continues, as the object's resize procedure should have been called by the geometry manager. If the geometry manager returns **XtGeometryNo**, **XtSetValues** ignores the geometry request and continues. If the geometry manager returns **XtGeometryAlmost**, **XtSetValues** calls the set_values_almost procedure, which determines what should be done. **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**, and the widget is realized, **XtSetValues** causes the widget's expose procedure to be invoked by calling **XClearArea** on the widget's window.

To modify the current values of resources associated with a widget instance using varargs lists, use **XtVaSetValues**.

void XtVaSetValues(*object*, ...)
    Widget *object*;

object Specifies the object whose resources are to be modified. Must be of class Object or any subclass thereof.

... Specifies the variable argument list of name/value pairs that contain the resources to be modified and their new values.

XtVaSetValues is identical in function to **XtSetValues** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1.

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, ArgList, Cardinal*);
    Widget current;
    Widget request;
    Widget new;
    ArgList args;
    Cardinal *num_args;
```

current Specifies a copy of the widget as it was before the **XtSetValues** call.

request 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.

new Specifies the widget with the new values that are actually allowed.

args Specifies the argument list passed to **XtSetValues** or the transformed argument list passed to **XtVaSetValues**.

num_args Specifies the number of entries in the argument list.

The set_values procedure should recompute any field derived from resources that are changed (for example, many GCs depend on foreground and background pixels). 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 a copy of the widget, updated as originally requested. The *new* widget starts with the values in the request, but it has additionally 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. The set_values procedure need not refer to the *request* widget unless it must resolve conflicts between the *current* and *new* widgets. Any changes the widget needs to make, including geometry changes, should be made in the *new* widget.

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. In this case, zero is a special value defined by the class to permit the application to invoke this behavior.

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. If the set_values procedure needs to call any routines that operate on a widget, it should specify *new* as the widget instance.

Before calling the set_values procedures, the Intrinsics modify the resources of the *request* widget according to the contents of the arglist; if the widget names all its resources in the class resource list, it is never necessary to examine the contents of *args*.

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 **XClearArea** 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 call to **XtSetValues**, its resize procedure is 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 procedure 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 the widget class record is of type **XtAlmostProc**.

typedef void (*XtAlmostProc)(Widget, Widget, XtWidgetGeometry*, XtWidgetGeometry*);
    Widget *old*;
    Widget *new*;
    XtWidgetGeometry **request*;
    XtWidgetGeometry **reply*;

old	Specifies a copy of the object as it was before the **XtSetValues** call.
new	Specifies the object instance record.
request	Specifies the original geometry request that was sent to the geometry manager that caused **XtGeometryAlmost** to be returned.
reply	Specifies the compromise geometry that was returned by the geometry manager with **XtGeometryAlmost**.

Most classes inherit the set_values_almost procedure from their superclass by specifying **XtInheritSetValuesAlmost** in the class initialization. The set_values_almost procedure in **rectObjClass** 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 **XtGeometryNo** or **XtGeometryAlmost** and a compromise geometry. The *new* object is the actual instance record. The *x*, *y*, *width*, *height*, and *border_width* fields contain the original values as they were before the **XtSetValues** call and all other fields contain the new values. The *request* parameter contains the new geometry request that was made to the parent. The *reply* parameter contains *reply->request_mode* equal to zero if the parent returned **XtGeometryNo** and contains the parent's compromise geometry otherwise. The set_values_almost procedure takes the original geometry and the compromise geometry and determines if the compromise is acceptable or whether to try a different compromise. It returns its results in the *request* parameter, which is then sent back to the geometry manager for another try. To accept the compromise, the procedure must copy the contents of the *reply* geometry into the *request* geometry; to attempt an alternative geometry, the procedure may modify any part of the *request* argument; to terminate the geometry negotiation and retain the original geometry, the procedure must set *request->request_mode* to zero. The geometry fields of the *old* and *new* instances must not be modified directly.

9.7.2.3. Widget State: the ConstraintClassPart 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 resources that are changed. Further, it may modify other 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 may reset the *height* field in the widget.

9.7.2.4. Widget Subpart State

To set the current values of subpart resources associated with a widget instance, use **XtSetSubvalues**. For a discussion of subpart resources, see Section 9.4.

void XtSetSubvalues(*base*, *resources*, *num_resources*, *args*, *num_args*)
    XtPointer *base*;
    XtResourceList *resources*;
    Cardinal *num_resources*;
    ArgList *args*;
    Cardinal *num_args*;

base	Specifies the base address of the subpart data structure into which the resources should be written.
resources	Specifies the subpart resource list.
num_resources	Specifies the number of entries in the resource list.
args	Specifies the argument list of name/value pairs that contain the resources to be modified and their new values.
num_args	Specifies the number of entries in the argument list.

The **XtSetSubvalues** function updates the resource fields of the structure identified by *base*. Any specified arguments that do not match an entry in the resource list are silently ignored.

To set the current values of subpart resources associated with a widget instance using varargs lists, use **XtVaSetSubvalues**.

void XtVaSetSubvalues(*base*, *resources*, *num_resources*, ...)
    XtPointer *base*;
    XtResourceList *resources*;
    Cardinal *num_resources*;

base	Specifies the base address of the subpart data structure into which the resources should be written.
resources	Specifies the subpart resource list.
num_resources	Specifies the number of entries in the resource list.
...	Specifies the variable argument list of name/value pairs that contain the resources to be modified and their new values.

XtVaSetSubvalues is identical in function to **XtSetSubvalues** with the *args* and *num_args* parameters replaced by a varargs list, as described in Section 2.5.1. **XtVaTypedArg** is not supported for **XtVaSetSubvalues**. If an entry containing **XtVaTypedArg** is specified in the list, a warning message is issued and the entry is ignored.

9.7.2.5. Widget Subpart Resource Data: the set_values_hook Procedure

Note

> The set_values_hook procedure is obsolete, as the same information is now available to the set_values procedure. The procedure has been retained for those widgets that used it in versions prior to Release 4.

Widgets that have a subpart can set the subpart resource values through **XtSetValues** by 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 subpart resource values are to be changed.

args Specifies the argument list that was passed to **XtSetValues** or the transformed varargs list passed to **XtVaSetValues**.

num_args Specifies the number of entries in the argument list.

The widget with subpart resources may call **XtSetValues** from the set_values_hook procedure and pass in its subresource list and the *args* and *num_args* parameters.

Chapter 10

Translation Management

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 sequences 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 resources XtNtranslations and XtNbaseTranslations are used to modify the class default translation table; see Section 10.3.

10.1. Action Tables

All widget class records contain an action table, an array of **XtActionsRec** entries. 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 directly. The translation action 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, or NULL.

num_params Specifies the number of entries in *params*.

```
typedef struct _XtActionsRec {
        String string;
        XtActionProc proc;
} XtActionsRec, *XtActionList;
```

The *string* field is the name used in translation tables to access the procedure. The *proc* field is a pointer to a procedure that implements the functionality.

When the action list is specified as the **CoreClassPart** *actions* field the string pointed to by *string* must be permanently allocated prior to or during the execution of the class initialization procedure and must not be subsequently deallocated.

Action procedures should not assume that the widget in which they are invoked is realized; an accelerator specification can cause an action procedure to be called for a widget that does not yet have a window. Widget writers should also note which of a widget's callback lists are invoked from action procedures and warn clients not to assume the widget is realized in those callbacks.

For example, a Pushbutton widget has procedures to take the following actions:

- Set the 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 Pushbutton widget class makes these functions available to translation tables written for Pushbutton or any subclass. The string entry is the name used in translation tables. The procedure entry (usually 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},
};
```

The Intrinsics reserve all action names and parameters starting with the characters ''Xt'' for future standard enhancements. Users, applications, and widgets should not declare action names or pass parameters starting with these characters except to invoke specified built-in Intrinsics functions.

10.1.1. Action Table Registration

The *actions* and *num_actions* fields of **CoreClassPart** specify the actions implemented by a widget class. These are automatically registered with the Intrinsics when the class is initialized and must be allocated in writable storage prior to Core class_part initialization, and never deallocated. To save memory and optimize access, the Intrinsics may overwrite the storage in order to compile the list into an internal representation.

To declare an action table within an application 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_actions 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 containing **XtMenuPopup** and **XtMenuPopdown** as part of **XtCreateApplicationContext**.

10.1.2. Action Names to Procedure Translations

The translation manager uses a simple algorithm to resolve 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, in order:

- The widget's class and all superclass action tables, in subclass-to-superclass order.
- The parent's class and all superclass action tables, in subclass-to-superclass order, then on up the ancestor tree.
- The action tables registered with **XtAppAddActions** and **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 a warning message.

10.1.3. Action Hook Registration

An application can specify a procedure that will be called just before every action routine is dispatched by the translation manager. To do so, the application supplies a procedure pointer of type **XtActionHookProc**.

typedef void (*XtActionHookProc)(Widget, XtPointer, String, XEvent*, String*, Cardinal*);
    Widget *w*;
    XtPointer *client_data*;
    String *action_name*;
    XEvent* *event*;
    String* *params*;
    Cardinal* *num_params*;

w	Specifies the widget whose action is about to be dispatched.
client_data	Specifies the application-specific closure that was passed to **XtAppAddActionHook.**
action_name	Specifies the name of the action to be dispatched.
event	Specifies the event argument that will be passed to the action routine.
params	Specifies the action parameters that will be passed to the action routine.
num_params	Specifies the number of entries in *params*.

Action hooks should not modify any of the data pointed to by the arguments other than the *client_data* argument.

To add an action hook, use **XtAppAddActionHook**.

XtActionHookId XtAppAddActionHook(*app*, *proc*, *client_data*)
    XtAppContext *app*;
    XtActionHookProc *proc*;
    XtPointer *client_data*;

app	Specifies the application context.
proc	Specifies the action hook procedure.
client_data	Specifies application-specific data to be passed to the action hook.

XtAppAddActionHook adds the specified procedure to the front of a list maintained in the application context. In the future, when an action routine is about to be invoked for any widget in this application context, either through the translation manager or via **XtCallActionProc**, the action hook procedures will be called in reverse order of registration just prior to invoking the action routine.

Action hook procedures are removed automatically and the **XtActionHookId**s destroyed when the application context in which they were added is destroyed.

To remove an action hook procedure without destroying the application context, use **XtRemoveActionHook**.

```
void XtRemoveActionHook(id)
    XtActionHookId id;
```

id Specifies the action hook id returned by **XtAppAddActionHook**.

XtRemoveActionHook removes the specified action hook procedure from the list in which it was registered.

10.2. Translation Tables

All widget instance records contain a translation table, which is a resource with a default value specified elsewhere in the class record. 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 Pushbutton 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 Pushbutton'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 **CoreClassPart** should be filled in at class 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*. In Core's class part initialization procedure, the Intrinsics compile this translation table into an efficient internal form. Then, at widget creation time, this default translation table is combined with the XtNtranslations and XtNbaseTranslations resources; see Section 10.3.

The resource conversion mechanism automatically compiles string translation tables that are specified in the resource database. If a client uses translation tables that are not retrieved via a resource conversion, 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, and an event-specific suffix.

Various abbreviations are supported to make translation tables easier to read. The events must match incoming events in left-to-right order to trigger the action sequence.

10.2.2. Action Sequences

Action sequences specify what program or widget actions to take in response to incoming X events. An action sequence consists of space-separated action procedure call specifications. Each action procedure call consists of the name of an action procedure and a parenthesized list of zero or more comma-separated string parameters to pass to that procedure. The actions are invoked in left-to-right order as specified in the action sequence.

10.2.3. Multi-click Time

Translation table entries may specify actions that are taken when two or more identical events occur consecutively within a short time interval, called the multi-click time. The multi-click time value may be specified as an application resource with name ‘‘multiClickTime’’ and class ‘‘MultiClickTime’’ and may also be modified dynamically by the application. The multi-click time is unique for each Display value and is retrieved from the resource database by **XtDisplayInitialize**. If no value is specified, the initial value is 200 milliseconds.

To set the multi-click time dynamically, use **XtSetMultiClickTime**.

void XtSetMultiClickTime(*display*, *time*)
    Display **display*;
    int *time*;

display Specifies the display connection.

time Specifies the multi-click time in milliseconds.

XtSetMultiClickTime sets the time interval used by the translation manager to determine when multiple events are interpreted as a repeated event. When a repeat count is specified in a translation entry, the interval between the timestamps in each pair of repeated events (e.g., between two **ButtonPress** events) must be less than the multi-click time in order for the translation actions to be taken.

To read the multi-click time, use **XtGetMultiClickTime**.

int XtGetMultiClickTime(*display*)
    Display **display*;

display Specifies the display connection.

XtGetMultiClickTime returns the time in milliseconds that the translation manager uses to determine if multiple events are to be interpreted as a repeated event for purposes of matching a translation entry containing a repeat count.

10.3. Translation Table Management

Sometimes an application needs to merge its own translations with a widget’s translations. For example, a window manager provides functions to move a window. The window manager wishes to bind this operation to a specific pointer button in the title bar without the possibility of user override and bind it to other buttons that may be overridden by the user.

To accomplish this, the window manager should first 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 (i.e., those that may be overridden). 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:

XtParseTranslationTable	Compiles a translation table.
XtAugmentTranslations	Merges a compiled translation table into a widget's compiled translation table, ignoring any new translations that conflict with existing translations.
XtOverrideTranslations	Merges a compiled translation table into a widget's compiled translation table, replacing any existing translations that conflict with new translations.

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, provided in the format given in Appendix B, into an 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 additional 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. Must be of class Core or any subclass thereof.

translations Specifies the compiled translation table to merge in.

The **XtAugmentTranslations** function merges the new translations into the existing widget translations, ignoring any **#replace**, **#augment**, or **#override** directive that may have been specified in the translation string. The translation table specified by *translations* is not altered by this process. **XtAugmentTranslations** logically appends the string representation of the new translations to the string representation of the widget's current translations and reparses the result with no warning messages about duplicate left-hand sides, then stores the result back into the widget instance; i.e., 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. Must be of class Core or any subclass thereof.

translations Specifies the compiled translation table to merge in.

The **XtOverrideTranslations** function merges the new translations into the existing widget translations, ignoring any **#replace**, **#augment**, or **#override** directive that may have been specified in the translation string. The translation table specified by *translations* is not altered by this process. **XtOverrideTranslations** logically appends the string representation of the widget's current translations to the string representation of the new translations and reparses the result with no warning messages about duplicate left-hand sides, then stores the result back into the widget instance; i.e., if the new translations contain an event or event sequence that already exists in the widget's translations, the new translation overrides 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 the string to specify whether the table should replace, augment, or override any existing translation table in the widget. To specify this, a sharp sign (#) is given as the first character of the table followed by one of the keywords ‘‘replace’’, ‘‘augment’’, or ‘‘override’’ to indicate whether to replace, augment, or override the existing table. The replace or merge operation is performed during the Core instance initialization and during the Core set_values invocation. Each merge operation produces a new translation resource value; if the original tables were shared by other widgets, they are unaffected. If no directive is specified, ‘‘#replace’’ is assumed.

At instance initialization the XtNtranslations resource is first fetched. Then, if it was not specified or did not contain ‘‘#replace’’, the resource database is searched for the resource XtNbaseTranslations. If XtNbaseTranslations is found it is merged into the widget class translation table. Then the widget *translations* field is merged into the result, or into the class translation table if XtNbaseTranslations was not found. This final table is then stored into the widget *translations* field. If the XtNtranslations resource specified ‘‘#replace’’ no merge is done. If neither XtNbaseTranslations or XtNtranslations are specified, the class translation table is copied into the widget instance.

To completely remove existing translations, use **XtUninstallTranslations**.

void XtUninstallTranslations(*w*)
  Widget *w*;

w Specifies the widget from which the translations are to be removed. Must be of class Core or any subclass thereof.

The **XtUninstallTranslations** function causes the entire translation table for the widget to be removed.

10.4. Using Accelerators

It is often desirable 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 lets you accomplish this. An accelerator table is a translation table that is bound with its actions in the context of a particular widget, the *source* widget. The accelerator table can then be installed on one or more *destination* widgets. When an event sequence in the destination widget would cause an accelerator action to be taken, and if the source widget is sensitive, the actions are executed as though triggered by the same event sequence in the accelerator source widget. The event is passed to the action procedure without modification. The action procedures used within accelerators must not assume that the source widget is realized nor that any fields of the event are in reference to the source widget's window if the widget is realized.

Each widget instance contains that widget's exported accelerator table as a resource. 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 source widget that supplied the accelerators.

string Specifies the string representation of the accelerators for this widget.

Accelerators can be specified in resource files, and the string representation is the same as for a translation table. However, the interpretation of the **#augment** and **#override** directives applies 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 an opaque internal representation. The client should set the XtNaccelerators resource of each widget that is to be activated by these translations to the returned value.

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. Must be of class Core or any subclass thereof.

source Specifies the widget from which the accelerators are to come. Must be of class Core or any subclass thereof.

The **XtInstallAccelerators** function installs the *accelerators* resource value from *source* onto *destination* by merging the the source accelerators into the destination translations. If the source *display_accelerator* field 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. Must be of class Core or any subclass thereof.

source Specifies the root widget of the widget tree from which the accelerators are to come. Must be of class Core or any subclass thereof.

The **XtInstallAllAccelerators** function recursively descends the widget tree rooted at *source* and installs the accelerators resource value 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 KeyCodes 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 Specifies a location in which to store a mask that indicates the subset of all modifiers that are examined by the key translator.

keysym_return Specifies a location in which to store 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.

The KeyCode-to-KeySym translator procedure must be implemented such that multiple calls with the same *display*, *keycode*, and *modifiers* return the same result until either a new case converter (**XtCaseProc**) is installed or a **MappingNotify** event is received.

The Intrinsics maintain tables internally to map KeyCodes to KeySyms for each open display. Translator procedures and other clients may share a single copy of this table to perform the same mapping.

To return a pointer to the KeySym-to-KeyCode mapping table for a particular display, use **XtGetKeysymTable**.

```
KeySym *XtGetKeysymTable(display, min_keycode_return, keysyms_per_keycode_return)
      Display *display;
      KeyCode *min_keycode_return;
      int *keysyms_per_keycode_return;
```

display Specifies the display whose table is required.

min_keycode_return
Returns the minimum KeyCode valid for the display.

keysyms_per_keycode_return
Returns the number of KeySyms stored for each KeyCode.

XtGetKeysymTable returns a pointer to the Intrinsics' copy of the server's KeyCode-to-KeySym table. This table must not be modified. There are *keysyms_per_keycode_return* KeySyms associated with each KeyCode, located in the table with indices starting at index

(test_keycode - min_keycode_return) * keysyms_per_keycode_return

for KeyCode *test_keycode*. Any entries that have no KeySyms associated with them contain the value **NoSymbol**. Clients should not cache the KeySym table but should call **XtGetKeysymTable** each time the value is needed, as the table may change prior to dispatching each event.

For more information on this table, see Section 12.7 in *Xlib – C Language X Interface*.

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 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 the Shift, Lock, and group modifiers with the interpretations defined in *X Window System Protocol*, Section 5. 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)(Display*, KeySym, KeySym*, KeySym*);
    Display **display*;
    KeySym *keysym*;
    KeySym **lower_return*;
    KeySym **upper_return*;

display Specifies the display connection for which the conversion is required.

keysym Specifies the KeySym to convert.

lower_return Specifies a location into which to store the lower-case equivalent for the KeySym.

upper_return Specifies a location into which to store the upper-case 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** 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 refresh the keyboard state if necessary. The default converter understands case conversion for all Latin KeySyms defined in *X Window System Protocol*, Appendix A.

To determine upper- and lower-case 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 lower-case equivalent of the KeySym.
upper_return	Returns the upper-case 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.

10.6. Obtaining a KeySym in an Action Procedure

When an action procedure is invoked on a **KeyPress** or **KeyRelease** event, it often has a need to retrieve the KeySym and modifiers corresponding to the event that caused it to be invoked. In order to avoid repeating the processing that was just performed by the Intrinsics to match the translation entry, the KeySym and modifiers are stored for the duration of the action procedure and are made available to the client.

To retrieve the KeySym and modifiers that matched the final event specification in the translation table entry, use **XtGetActionKeysym**.

KeySym XtGetActionKeysym(*event*, *modifiers_return*)
  XEvent **event*;
  Modifiers **modifiers_return*;

event	Specifies the event pointer passed to the action procedure by the Intrinsics.
modifiers_return	Returns the modifiers that caused the match, if non-NULL.

If **XtGetActionKeysym** is called after an action procedure has been invoked by the Intrinsics and before that action procedure returns, and if the event pointer has the same value as the event pointer passed to that action routine, and if the event is a **KeyPress** or **KeyRelease** event, then **XtGetActionKeysym** returns the KeySym that matched the final event specification in the translation table and, if *modifiers_return* is non-NULL, the modifier state actually used to generate this KeySym; otherwise, if the event is a **KeyPress** or **KeyRelease** event, then **XtGetActionKeysym** calls **XtTranslateKeycode** and returns the results; else it returns **NoSymbol** and does not examine *modifiers_return*.

Note that if an action procedure invoked by the Intrinsics invokes a subsequent action procedure (and so on) via **XtCallActionProc**, the nested action procedure may also call **XtGetActionKeysym** to retrieve the Intrinsics' KeySym and modifiers.

10.7. KeySym-to-KeyCode Conversions

To return the list of KeyCodes that map to a particular KeySym in the keyboard mapping table maintained by the Intrinsics, use **XtKeysymToKeycodeList**.

void XtKeysymToKeycodeList(*display*, *keysym*, *keycodes_return*, *keycount_return*)
    Display **display*;
    KeySym *keysym*;
    KeyCode ***keycodes_return*;
    Cardinal **keycount_return*;

display	Specifies the display whose table is required.
keysym	Specifies the KeySym for which to search.
keycodes_return	Returns a list of KeyCodes that have *keysym* associated with them, or NULL if *keycount_return* is 0.
keycount_return	Returns the number of KeyCodes in the keycode list.

The **XtKeysymToKeycodeList** procedure returns all the KeyCodes that have *keysym* in their entry for the keyboard mapping table associated with *display*. For each entry in the table, the first four KeySyms (groups 1 and 2) are interpreted as specified by *X Window System Protocol*, Section 5. If no KeyCodes map to the specified KeySym, *keycount_return* is zero and **keycodes_return* is NULL.

The caller should free the storage pointed to by *keycodes_return* using **XtFree** when it is no longer useful. If the caller needs to examine the KeyCode-to-KeySym table for a particular KeyCode, it should call **XtGetKeysymTable**.

10.8. Registering Button and Key Grabs For Actions

To register button and key grabs for a widget's window according to the event bindings in the widget's translation table, use **XtRegisterGrabAction**.

void XtRegisterGrabAction(*action_proc*, *owner_events*, *event_mask*, *pointer_mode*, *keyboard_mode*)
    XtActionProc *action_proc*;
    Boolean *owner_events*;
    unsigned int *event_mask*;
    int *pointer_mode*, *keyboard_mode*;

action_proc Specifies the action procedure to search for in translation tables.

owner_events
event_mask
pointer_mode
keyboard_mode Specify arguments to **XtGrabButton** or **XtGrabKey**.

XtRegisterGrabAction adds the specified *action_proc* to a list known to the translation manager. When a widget is realized, or when the translations of a realized widget or the accelerators installed on a realized widget are modified, its translation table and any installed accelerators are scanned for action procedures on this list. If any are invoked on **ButtonPress** or **KeyPress**

events as the only or final event in a sequence, the Intrinsics will call **XtGrabButton** or **XtGrabKey** for the widget with every button or KeyCode which maps to the event detail field, passing the specified *owner_events*, *event_mask*, *pointer_mode*, and *keyboard_mode*. For **ButtonPress** events, the modifiers specified in the grab are determined directly from the translation specification and *confine_to* and *cursor* are specified as **None**. For **KeyPress** events, if the translation table entry specifies colon (:) in the modifier list, the modifiers are determined by calling the key translator procedure registered for the display and calling **XtGrabKey** for every combination of standard modifiers which map the KeyCode to the specified event detail KeySym, and ORing any modifiers specified in the translation table entry, and *event_mask* is ignored. If the translation table entry does not specify colon in the modifier list, the modifiers specified in the grab are those specified in the translation table entry only. For both **ButtonPress** and **KeyPress** events, don't-care modifiers are ignored unless the translation entry explicitly specifies ''Any'' in the *modifiers* field.

If the specified *action_proc* is already registered for the calling process, the new values will replace the previously specified values for any widgets that become realized following the call, but existing grabs are not altered on currently-realized widgets.

When translations or installed accelerators are modified for a realized widget, any previous key or button grabs registered as a result of the old bindings are released if they do not appear in the new bindings and are not explicitly grabbed by the client with **XtGrabKey** or **XtGrabButton**.

10.9. Invoking Actions Directly

Normally action procedures are invoked by the Intrinsics when an event or event sequence arrives for a widget. To invoke an action procedure directly, without generating (or synthesizing) events, use **XtCallActionProc**.

void XtCallActionProc(*widget*, *action*, *event*, *params*, *num_params*)
    Widget *widget*;
    String *action*;
    XEvent **event*;
    String **params*;
    Cardinal *num_params*;

widget	Specifies the widget in which the action is to be invoked. Must be of class Core or any subclass thereof.
action	Specifies the name of the action routine.
event	Specifies the contents of the *event* passed to the action routine.
params	Specifies the contents of the *params* passed to the action routine.
num_params	Specifies the number of entries in *params*.

XtCallActionProc searches for the named action routine in the same manner and order as translation tables are bound, as described in Section 10.1.2, except that application action tables are searched, if necessary, as of the time of the call to **XtCallActionProc**. If found, the action routine is invoked with the specified widget, event pointer, and parameters. It is the responsibility of the caller to ensure that the contents of the *event*, *params*, and *num_params* arguments are appropriate for the specified action routine and, if necessary, that the specified widget is realized or sensitive. If the named action routine cannot be found, **XtCallActionProc** generates a warning message and returns.

10.10. Obtaining a Widget's Action List

Occasionally a subclass will require the pointers to one or more of its superclass's action procedures. This would be needed, for example, in order to envelope the superclass's action. To retrieve the list of action procedures registered in the superclass's *actions* field, use

XtGetActionList.

void XtGetActionList(*widget_class*, *actions_return*, *num_actions_return*)
    WidgetClass *widget_class*;
    XtActionList **actions_return*;
    Cardinal **num_actions_return*;

widget_class Specifies the widget class whose actions are to be returned.

actions_return Returns the action list.

num_actions_return
Returns the number of action procedures declared by the class.

XtGetActionList returns the action table defined by the specified widget class. This table does not include actions defined by the superclasses. If *widget_class* is not initialized, or is not **coreWidgetClass** or a subclass thereof, or if the class does not define any actions, **actions_return* will be NULL and **num_actions_return* will be zero. If **actions_return* is non-NULL the client is responsible for freeing the table using **XtFree** when it is no longer needed.

Chapter 11

Utility Functions

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.
- Locate a widget given a window id.
- Handle errors.
- Set the WM_COLORMAP_WINDOWS property.
- Locate files by name with string substitutions.

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*)
    ArrayType array;

array Specifies a fixed-size array of arbitrary type.

The **XtNumber** macro returns the number of elements allocated to the array.

11.2. Translating Strings to Widget Instances

To translate a widget name to a widget instance, use **XtNameToWidget**.

Widget XtNameToWidget(*reference*, *names*)
    Widget *reference*;
    String *names*;

reference Specifies the widget from which the search is to start. Must be of class Core or any subclass thereof.

names Specifies the partially qualified name of the desired widget.

The **XtNameToWidget** function searches for a descendant of the *reference* widget whose name matches the specified names. The *names* parameter specifies a simple object name or a series of simple object name components separated by periods or asterisks. **XtNameToWidget** returns the descendant with the shortest name matching the specification according to the following rules, where child is either a pop-up child or a normal child if the widget's class is a subclass of Composite :

- Enumerate the object subtree rooted at the reference widget in breadth-first order, qualifying the name of each object with the names of all its ancestors up to but not including the reference widget. The ordering between children of a common parent is not defined.

- Return the first object in the enumeration that matches the specified name, where each component of *names* matches exactly the corresponding component of the qualified object name, and asterisk matches any series of components, including none.
- If no match is found, return NULL.

Since breadth-first traversal is specified, the descendant with the shortest matching name (i.e., the fewest number of components), if any, will always be returned. However, since the order of enumeration of children is undefined and since the Intrinsics do not require that all children of a widget have unique names, **XtNameToWidget** may return any child that matches if there are multiple objects in the subtree with the same name. Consecutive separators (periods or asterisks) including at least one asterisk are treated as a single asterisk. Consecutive periods are treated as a single period.

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** function 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 each 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**. **XtCalloc** returns the address of the allocated storage.

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 allocated with **XtMalloc**, **XtCalloc**, or **XtRealloc**, or NULL.

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** simply calls **XtMalloc**. **XtRealloc** then returns the address of the new block.

To free an allocated block of storage, use **XtFree**.

void XtFree(*ptr*)
    char **ptr*;

ptr Specifies a pointer to a block of storage allocated with **XtMalloc**, **XtCalloc**, or **XtRealloc**, or NULL.

The **XtFree** function returns storage, allowing it to be reused. If *ptr* is NULL, **XtFree** returns immediately.

To allocate storage for a new instance of a type, use **XtNew**.

type *XtNew(*type*)
    type t;

type Specifies a previously declared 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)))
```

The storage allocated by **XtNew** should be freed using **XtFree**.

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))
```

The storage allocated by **XtNewString** should be freed using **XtFree**.

11.4. Sharing Graphics Contexts

The Intrinsics provide a mechanism whereby cooperating objects 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 allows for clients to declare both modifiable and nonmodifiable fields of the shared GCs.

To obtain a shareable GC with modifiable fields, use **XtAllocateGC**.

GC XtAllocateGC(*widget*, *depth*, *value_mask*, *values*, *dynamic_mask*, *unused_mask*)
    Widget *object*;
    Cardinal *depth*;
    XtGCMask *value_mask*;
    XGCValues **values*;
    XtGCMask *dynamic_mask*;
    XtGCMask *unused_mask*;

object	Specifies an object, giving the screen for which the returned GC is valid. Must be of class Object or any subclass thereof.
depth	Specifies the depth for which the returned GC is valid, or 0.
value_mask	Specifies fields of the GC that are initialized from *values*.
values	Specifies the values for the initialized fields.
dynamic_mask	Specifies fields of the GC that may be modified by the caller.
unused_mask	Specifies fields of the GC that will not be used by the caller.

The **XtAllocateGC** function returns a shareable GC that may be modified by the client. The *screen* field of the specified widget or of the nearest widget ancestor of the specified object and the specified *depth* argument supply the root and drawable depths for which the GC is to be valid. If *depth* is zero the depth is taken from the *depth* field of the specified widget or of the nearest widget ancestor of the specified object.

The *value_mask* argument specifies fields of the GC that will be initialized with the respective member of the *values* structure. The *dynamic_mask* argument specifies fields that the caller intends to modify during program execution. The caller must insure that the corresponding GC field is set prior to each use of the GC. The *unused_mask* argument specifies fields of the GC that are of no interest to the caller. The caller may make no assumptions about the contents of any fields specified in *unused_mask*. The caller may assume that at all times all fields not specified in either *dynamic_mask* or *unused_mask* have their default value if not specified in *value_mask* or the value specified by *values*. If a field is specified in both *value_mask* and *dynamic_mask*, the effect is as if it were specified only in *dynamic_mask* and then immediately set to the value in *values*. If a field is set in *unused_mask* and also in either *value_mask* or *dynamic_mask*, the specification in *unused_mask* is ignored.

XtAllocateGC tries to minimize the number of unique GCs created by comparing the arguments with those of previous calls and returning an existing GC when there are no conflicts. **XtAllocateGC** may modify and return an existing GC if it was allocated with a nonzero *unused_mask*.

To obtain a shareable GC with no modifiable fields, use **XtGetGC**.

GC XtGetGC(*object*, *value_mask*, *values*)
    Widget *object*;
    XtGCMask *value_mask*;
    XGCValues **values*;

object	Specifies an object, giving the screen and depth for which the returned GC is valid. Must be of class Object or any subclass thereof.
value_mask	Specifies which fields of the *values* structure are specified.
values	Specifies the actual values for this GC.

The **XtGetGC** function returns a shareable, read-only GC. The parameters to this function are the same as those for **XCreateGC** except that an Object is passed instead of a Display.

XtGetGC is equivalent to **XtAllocateGC** with *depth*, *dynamic_mask*, and *unused_mask* all zero.

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 from *values* and which it should fill in with default values.

To deallocate a shared GC when it is no longer needed, use **XtReleaseGC**.

void XtReleaseGC(*object*, *gc*)
    Widget *object*;
    GC *gc*;

object Specifies any object on the Display for which the GC was created. Must be of class Object or any subclass thereof.

gc Specifies the shared GC obtained with either **XtAllocateGC** or **XtGetGC**.

References to shareable GCs are counted and a free request is generated to the server when the last user of a given GC releases it.

11.5. Managing Selections

Arbitrary widgets in multiple applications can communicate with each other by means of the Intrinsics global selection mechanism, which conforms to the specifications in the *Inter-Client Communication Conventions Manual*. The Intrinsics supply functions for providing and receiving selection data in one logical piece (atomic transfers) or in smaller logical segments (incremental transfers).

The incremental interface is provided for a selection owner or selection requestor that cannot or prefers not to pass the selection value to and from the Intrinsics in a single call. For instance, either an application that is running on a machine with limited memory may not be able to store the entire selection value in memory, 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 additional storage to copy the pieces contiguously. Any owner or requestor that prefers to deal with the selection value in segments can use the incremental interfaces to do so. The transfer between the selection owner or requestor and the Intrinsics is not required to match the underlying transport protocol between the application and the X server; the Intrinsics will break a too large selection into smaller pieces for transport if necessary and will coalesce a selection transmitted incrementally if the value was requested atomically.

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 as retrieved by **XtDisplayInitialize**. If selectionTimeout is not specified, the default is five seconds.

11.5.2. Using Atomic Transfers

When using atomic transfers, the owner will completely process one selection request at a time. The owner may consider each request individually, since there is no possibility for overlap between evaluation of two requests.

11.5.2.1. Atomic Transfer Procedures

The following procedures are used by the selection owner when providing selection data in a single unit.

The procedure pointer specified by the owner to supply the selection data to the Intrinsics is of type **XtConvertSelectionProc**.

```
typedef Boolean (*XtConvertSelectionProc)(Widget, Atom*, Atom*, Atom*,
                XtPointer*, unsigned long*, int*);
   Widget w;
   Atom *selection;
   Atom *target;
   Atom *type_return;
   XtPointer *value_return;
   unsigned long *length_return;
   int *format_return;
```

w	Specifies the widget that currently owns this selection.
selection	Specifies the atom naming the 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 Intrinsics 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_return*, each of size indicated by *format_return*, 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 Intrinsics 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 into which it is prepared to convert the selection. The value returned in *format_return* must be one of 8, 16, or 32 to allow the server to byte-swap the data if necessary.

This procedure does not need to worry about responding to the MULTIPLE or the TIMESTAMP target values (see Section 2.6.2 in the *Inter-Client Communication Conventions Manual*). A selection request with the MULTIPLE target type will be transparently transformed into a series of calls to this procedure, one for each target type, and a selection request with the TIMESTAMP target value will be answered automatically by the Intrinsics using the time specified in the call to **XtOwnSelection** or **XtOwnSelectionIncremental**.

To retrieve the **SelectionRequest** event that triggered the **XtConvertSelectionProc** procedure, use **XtGetSelectionRequest**.

XSelectionRequestEvent *XtGetSelectionRequest(*w*, *selection*, *request_id*)
    Widget *w*;
    Atom *selection*;
    XtRequestId *request_id*;

w	Specifies the widget that currently owns this selection. Must be of class Core or any subclass thereof.
selection	Specifies the selection being processed.
request_id	Specifies the requestor id in the case of incremental selections, or NULL in the case of atomic transfers.

XtGetSelectionRequest may only be called from within an **XtConvertSelectionProc** procedure and returns a pointer to the **SelectionRequest** event that caused the conversion procedure to be invoked. *Request_id* specifies a unique id for the individual request in the case that multiple incremental transfers are outstanding. For atomic transfers, *request_id* must be specified as NULL. If no **SelectionRequest** event is being processed for the specified *widget*, *selection*, and *request_id*, **XtGetSelectionRequest** returns NULL.

The procedure pointer specified by the owner when it desires notification upon losing ownership is of type **XtLoseSelectionProc**.

typedef void (*XtLoseSelectionProc)(Widget, Atom*);
   Widget *w*;
   Atom **selection*;

w	Specifies the widget that has lost selection ownership.
selection	Specifies the atom naming the selection.

This procedure is called by the Intrinsics selection mechanism to inform the specified widget that it has lost the given selection. Note that this procedure does not ask the widget to relinquish the selection ownership; it is merely informative.

The procedure pointer specified by the owner when it desires notification of receipt of the data or when it manages the storage containing the data is of type **XtSelectionDoneProc**.

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 naming the selection that was converted.

target Specifies the target type to which the conversion was done.

This procedure is called by the Intrinsics selection mechanism to inform the selection owner that 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, 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.

11.5.2.2. Getting the Selection Value

The procedure pointer specified by the requestor to receive the selection data from the Intrinsics is of type **XtSelectionCallbackProc**.

```
typedef void (*XtSelectionCallbackProc)(Widget, XtPointer, Atom*, Atom*, XtPointer, unsigned long*, int*);
    Widget w;
    XtPointer client_data;
    Atom *selection;
    Atom *type;
    XtPointer 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 name of the 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 (which the client must remember for itself) but the type that is used to represent the target. The special symbolic constant **XT_CONVERT_FAIL** is used to indicate that the selection conversion failed because the selection owner did not respond within the Intrinsics 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.

If the **SelectionNotify** event returns a property of **None**, meaning the conversion has been refused because there is no owner for the specified selection or the owner cannot convert the selection to the requested target for any reason, the procedure is called with a value of NULL and a length of zero.

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;
    XtPointer client_data;
    Time time;
```

w Specifies the widget making the request. Must be of class Core or any subclass thereof.

selection Specifies the particular selection desired; for example, **XA_PRIMARY**.

target Specifies the type of information needed about the selection.

callback Specifies the procedure to be called when the selection value has been obtained. Note that this is how the selection value is communicated back to the client.

client_data Specifies additional data to be passed to the specified procedure when it is called.

time Specifies the timestamp that indicates when the selection request was initiated. This should be the timestamp of the event that triggered this request; the value **CurrentTime** is not acceptable.

The **XtGetSelectionValue** function requests the value of the selection converted to the target type. The specified callback will be called at some time after **XtGetSelectionValue** is called, when the selection data is received from the X server. It may be called before or after **XtGetSelectionValue** returns. For more information about *selection*, *target*, and *time*, see Section 2.6 in the *Inter-Client Communication Conventions Manual*.

```
void XtGetSelectionValues(w, selection, targets, count, callback, client_data, time)
    Widget w;
    Atom selection;
    Atom *targets;
    int count;
    XtSelectionCallbackProc callback;
    XtPointer *client_data;
    Time time;
```

w Specifies the widget making the request. Must be of class Core or any subclass thereof.

selection Specifies the particular selection desired (that is, primary or secondary).

targets Specifies the types of information needed about the selection.

count Specifies the length of *the* targets and *client_data* lists.

callback Specifies the callback procedure to be called with each selection value obtained. Note that this is how the selection values are communicated back to the client.

client_data Specifies a list of additional data values, one for each target type, that are passed to the callback procedure when it is called for that target.

time Specifies the timestamp that indicates when the selection request was initiated. This should be the timestamp of the event that triggered this request; the value **CurrentTime** is not acceptable.

The **XtGetSelectionValues** function is similar to multiple calls to **XtGetSelectionValue** except that it guarantees that no other client can assert ownership between requests and therefore that all the conversions will refer to the same selection value. The callback is invoked once for each target value with the corresponding client data. For more information about *selection*, *target*, and *time* see Section 2.6 in the *Inter-Client Communication Conventions Manual*.

11.5.2.3. Setting the Selection Owner

To set the selection owner and indicate that the selection value will be provided in one piece, 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. Must be of class Core or any subclass thereof.

selection Specifies the name of the selection (for example, **XA_PRIMARY**).

time Specifies the timestamp that indicates when the ownership request was initiated. This should be the timestamp of the event that triggered ownership; the value **CurrentTime** is not acceptable.

convert_proc Specifies the procedure to be called whenever a client requests the current value of the selection.

lose_selection Specifies the procedure 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 called after the requestor has received the selection value, or NULL if the owner is not interested in being called back.

The **XtOwnSelection** function informs the Intrinsics selection mechanism that a widget wishes to own a selection. It returns **True** if the widget successfully becomes 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. The widget can lose selection ownership either because some other client asserted later ownership of the selection or because the widget voluntarily gave up ownership of the selection. The lose_selection procedure is not called if the widget fails to obtain selection ownership in the first place.

If a done_proc is specified, the client owns the storage allocated for passing the value to the Intrinsics. If *done_proc* is NULL, the convert_proc must allocate storage using **XtMalloc**, **XtRealloc**, or **XtCalloc**, and the value specified will be freed by the Intrinsics when the transfer is complete.

Usually, a selection owner maintains ownership indefinitely until some other client requests ownership, at which time the Intrinsics selection mechanism informs the previous owner that it has lost ownership of the selection. However, in response to some user actions (for example, when a user deletes the information selected), the application may with to explicitly inform the Intrinsics that it no longer is to be the selection owner by using **XtDisownSelection**.

void XtDisownSelection(*w*, *selection*, *time*)
    Widget *w*;
    Atom *selection*;
    Time *time*;

w Specifies the widget that wishes to relinquish ownership.

selection Specifies the atom naming the selection being given up.

time Specifies the timestamp that indicates when the request to relinquish selection ownership was initiated.

The **XtDisownSelection** function informs the Intrinsics 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 process more than one selection request for the same selection, converted to the same target, at the same time. The incremental functions take a *request_id* 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 a different requestor for the same selection value.
- To distinguish between the requests, the owner uses the request_id value. This allows the owner to distinguish between the first requestor, which is asking for the second segment, and the second requestor, which is asking for the first segment.

11.5.3.1. Incremental Transfer Procedures

The following procedures are used by selection owners who wish to provide the selection data in multiple segments.

The procedure pointer specified by the incremental owner to supply the selection data to the Intrinsics is of type **XtConvertSelectionIncrProc**.

```
typedef XtPointer XtRequestId;

typedef Boolean (*XtConvertSelectionIncrProc)(Widget, Atom*, Atom*, Atom*, XtPointer*,
                    unsigned long*, int*, unsigned long*, XtPointer, XtRequestId*);
    Widget w;
    Atom *selection;
    Atom *target;
    Atom *type_return;
    XtPointer *value_return;
    unsigned long *length_return;
    int *format_return;
    unsigned long *max_length;
    XtPointer client_data;
    XtRequestId *request_id;
```

w	Specifies the widget that currently owns this selection.
selection	Specifies the atom that names the selection requested.
target	Specifies the type of information required about the selection.
type_return	Specifies a pointer to an atom into which the property type of the converted value of the selection is to be stored.
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.
length_return	Specifies a pointer into which the number of elements in *value_return*, each of size indicated by *format_return*, 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 so that the server may byte-swap the data if necessary.

max_length Specifies the maximum number of bytes which may be transferred at any one time.

client_data Specifies the value passed in by the widget when it took ownership of the selection.

request_id Specifies an opaque identification for a specific request.

This procedure is called repeatedly by the Intrinsics selection mechanism to get the next incremental chunk of data from a selection owner who has called **XtOwnSelectionIncremental**. It must return **True** if the procedure has succeeded in converting the selection data or **False** otherwise. On the first call with a particular request id, the owner must begin a new incremental transfer for the requested selection and target. On subsequent calls with the same request id, the owner may assume that the previously supplied value is no longer needed by the Intrinsics; that is, a fixed transfer area may be allocated and returned in *value_return* for each segment to be transferred. This procedure should store a non-NULL value in *value_return* and zero in *length_return* to indicate that the entire selection has been delivered. After returning this final segment, the request id may be reused by the Intrinsics to begin a new transfer.

To retrieve the **SelectionRequest** event that triggered the selection conversion procedure, use **XtGetSelectionRequest**, described in Section 11.5.2.1.

The procedure pointer specified by the incremental selection owner when it desires notification upon no longer having ownership is of type **XtLoseSelectionIncrProc**.

typedef void (*XtLoseSelectionIncrProc)(Widget, Atom*, XtPointer);
    Widget *w*;
    Atom **selection*;
    XtPointer *client_data*;

w Specifies the widget that has lost the selection ownership.

selection Specifies the atom that names the selection.

client_data 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.

The procedure pointer specified by the incremental selection owner when it desires notification of receipt of the data or when it manages the storage containing the data is of type **XtSelectionDoneIncrProc**.

typedef void (*XtSelectionDoneIncrProc)(Widget, Atom*, Atom*, XtRequestId*, XtPointer);
    Widget *w*;
    Atom **selection*;
    Atom **target*;
    XtRequestId **request_id*;
    XtPointer *client_data*;

w Specifies the widget that owns the selection.

selection Specifies the atom that names the selection being transferred.

target Specifies the target type to which the conversion was done.

request_id Specifies an opaque identification for a specific request.

client_data Specified the value passed in by the widget when it took ownership of the selection.

This procedure, which is optional, is called by the Intrinsics after the requestor has retrieved the final (zero-length) segment of the incremental transfer to indicate that the entire transfer is complete. If this procedure is not specified, the Intrinsics will free only the final value returned by the selection owner using **XtFree**.

The procedure pointer specified by the incremental selection owner to notify it if a transfer should be terminated prematurely is of type **XtCancelConvertSelectionProc**.

typedef void (*XtCancelConvertSelectionProc)(Widget, Atom*, Atom*, XtRequestId*, XtPointer);
    Widget *w*;
    Atom **selection*;
    Atom **target*;
    XtRequestId **request_id*;
    XtPointer *client_data*;

w	Specifies the widget that owns the selection.
selection	Specifies the atom that names the selection being transferred.
target	Specifies the target type to which the conversion was done.
request_id	Specifies an opaque identification for a specific request.
client_data	Specifies the value passed in by the widget when it took ownership of the selection.

This procedure is called by the Intrinsics when it has been determined by means of a timeout or other mechanism that any remaining segments of the selection no longer need to be transferred. Upon receiving this callback, the selection request is considered complete and the owner can free the memory and any other resources that have been allocated for the transfer.

11.5.3.2. Getting the Selection Value Incrementally

To obtain the value of the selection using incremental transfers, use **XtGetSelectionValueIncremental** or **XtGetSelectionValuesIncremental**.

void XtGetSelectionValueIncremental(*w*, *selection*, *target*, *selection_callback*, *client_data*, *time*)
    Widget *w*;
    Atom *selection*;
    Atom *target*;
    XtSelectionCallbackProc *selection_callback*;
    XtPointer *client_data*;
    Time *time*;

w	Specifies the widget making the request. Must be of class Core or any subclass thereof.
selection	Specifies the particular selection desired.
target	Specifies the type of information needed about the selection.
selection_callback	Specifies the callback procedure to be called to receive each data segment.
client_data	Specifies client-specific data to be passed to the specified callback procedure when it is invoked.
time	Specifies the timestamp that indicates when the selection request was initiated. This should be the timestamp of the event that triggered this request; the value **CurrentTime** is not acceptable.

The **XtGetSelectionValueIncremental** function is similar to **XtGetSelectionValue** except that the selection_callback procedure will be called repeatedly upon delivery of multiple segments of the selection value. The end of the selection value is indicated when *selection_callback* is called with a non-NULL value of length zero, which must still be freed by the client. If the transfer of the selection is aborted in the middle of a transfer (for example, because to timeout), the selection_callback procedure is called with a type value equal to the symbolic constant **XT_CONVERT_FAIL** so that the requestor can dispose of the partial selection value it has collected up until that point. Upon receiving **XT_CONVERT_FAIL**, the requesting client must determine for itself whether or not a partially completed data transfer is meaningful. For more information about *selection*, *target*, and *time*, see Section 2.6 in the *Inter-Client Communication Conventions Manual*.

void XtGetSelectionValuesIncremental(*w*, *selection*, *targets*, *count*, *selection_callback*, *client_data*, *time*)
    Widget *w*;
    Atom *selection*;
    Atom **targets*;
    int *count*;
    XtSelectionCallbackProc *selection_callback*;
    XtPointer **client_data*;
    Time *time*;

w Specifies the widget making the request. Must be of class Core or any subclass thereof.

selection Specifies the particular selection desired.

targets Specifies the types of information needed about the selection.

count Specifies the length of the *targets* and *client_data* lists.

selection_callback
Specifies the callback procedure to be called to receive each selection value.

client_data Specifies a list of client data (one for each target type) values that are passed to the callback procedure when it is invoked for the corresponding target.

time Specifies the timestamp that indicates when the selection request was initiated. This should be the timestamp of the event that 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 except that **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. For more information about *selection*, *target*, and *time*, see Section 2.6 in the *Inter-Client Communication Conventions Manual*.

11.5.3.3. Setting the Selection Owner for Incremental Transfers

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*;
XtPointer *client_data*;

w Specifies the widget that wishes to become the owner. Must be of class Core or any subclass thereof.

selection Specifies the atom that names the selection.

time Specifies the timestamp that indicates when the selection ownership request was initiated. This should be the timestamp of the event that triggered ownership; the value **CurrentTime** is not acceptable.

convert_callback Specifies the procedure to be called whenever the current value of the selection is requested.

lose_callback Specifies the procedure to be called whenever the widget has lost selection ownership, or NULL if the owner is not interested in being notified.

done_callback Specifies the procedure called after the requestor has received the entire selection, or NULL if the owner is not interested in being notified.

cancel_callback Specifies the callback procedure to be called when a selection request aborts because a timeout expires, or NULL if the owner is not interested in being notified.

client_data Specifies the argument to be passed to each of the callback procedures when they are called.

The **XtOwnSelectionIncremental** procedure informs the Intrinsics incremental selection mechanism that the specified widget wishes to own the selection. It returns **True** if the specified widget successfully becomes the selection owner or **False** otherwise. For more information about *selection*, *target*, and *time*, see Section 2.6 in the *Inter-Client Communication Conventions Manual*.

If a done_callback procedure is specified, the client owns the storage allocated for passing the value to the Intrinsics. If *done_callback* is NULL, the convert_callback procedure must allocate storage using **XtMalloc**, **XtRealloc**, or **XtCalloc**, and the final value specified will be freed by the Intrinsics when the transfer is complete. After a selection transfer has started, only one of the done_callback or cancel_callback procedures will be invoked to indicate completion of the transfer.

The lose_callback procedure does not indicate completion of any in-progress transfers; it will be invoked at the time a **SelectionClear** event is dispatched regardless of any active transfers, which are still expected to continue.

A widget that becomes the selection owner using **XtOwnSelectionIncremental** may use **XtDisownSelection** to relinquish selection ownership.

11.5.4. Retrieving the Most Recent Timestamp

To retrieve the timestamp from the most recent call to **XtDispatchEvent** that contained a timestamp, use **XtLastTimestampProcessed**.

Time XtLastTimestampProcessed(*display*)
    Display **display*;

display Specifies an open display connection.

If no **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **MotionNotify**, **EnterNotify**, **LeaveNotify**, **PropertyNotify**, or **SelectionClear** event has yet been passed to **XtDispatchEvent** for the specified display, **XtLastTimestampProcessed** returns zero.

11.6. Merging Exposure Events into a Region

The Intrinsics provide an **XtAddExposureToRegion** utility function that merges **Expose** and **GraphicsExpose** events into a region for clients to process at once rather than processing individual rectangles. For further information about regions, see Section 16.5 in *Xlib – C Language X Interface*.

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 window absolute 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. Must be of class RectObj or any subclass thereof.

x
y Specify the widget-relative x and y coordinates.

rootx_return
rooty_return Return 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 given 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 Specifies the window for which you want the widget.

If there is a realized widget whose window is the specified *window* on the specified *display*, **XtWindowToWidget** returns that widget; otherwise, it returns NULL.

11.9. Handling Errors

The Intrinsics allow a client to register procedures that will 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 retrieves the error message text from an error resource database.
- A low-level interface that takes a simple string to display.

The high-level functions construct a string to pass to the lower-level interface. The strings may be specified in application code and will be overridden by the contents of an external system-wide file, the ‘‘error database file’’. The location and name of this file is implementation dependent.

Note

> The application-context-specific error handling is not implemented on many systems, although the interfaces are always present. Most implementations will have just one set of error handlers for all application contexts within a process. If they are set for different application contexts, the ones registered 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 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 to be concatenated with the specified type to form the resource name of the error message.

type Specifies the type to be 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 parameters to be substituted in the message.

num_params Specifies the number of entries in *params*.

The specified name can be a general kind of error, like ‘‘invalidParameters’’ or ‘‘invalidWindow’’, and the specified type gives extra information such as the name of the routine in which the error was detected. 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;
    String name, type, class;
    String default;
    String buffer_return;
    int nbytes;
    XrmDatabase database;
```

app_context Specifies the application context.

name
type Specify the name and type 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 to be used, or NULL if the application context’s error 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 form the full resource name and class when querying the database, the *name* and *type* are concatenated with a single ‘‘.’’ between them and the *class* is concatenated with itself with a single ‘‘.’’ if it does not already contain a ‘‘.’’.

To return the application name and class as passed to **XtDisplayInitialize** for a particular Display, use **XtGetApplicationNameAndClass**.

```
void XtGetApplicationNameAndClass(display, name_return, class_return)
    Display* display;
    String* name_return;
    String* class_return;
```

display Specifies an open display connection that has been initialized with **XtDisplayInitialize**.

name_return Returns the application name.

class_return Returns the application class.

XtGetApplicationNameAndClass returns the application name and class passed to **XtDisplay-Initialize** for the specified display. If the display was never initialized or has been closed, the result is undefined. The returned strings are owned by the Intrinsics and must not be modified or freed by the caller.

To register a procedure to be called on fatal error conditions, use **XtAppSetErrorMsgHandler**.

XtErrorMsgHandler 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.

XtAppSetErrorMsgHandler returns a pointer to the previously installed high-level fatal error handler. The default high-level fatal error handler provided by the Intrinsics is named **_XtDe-faultErrorMsg** and constructs a string from the error resource database and calls **XtError**. Fatal error message handlers should not return. If one does, subsequent Intrinsics 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 entries in *params*.

The Intrinsics internal errors all have class ‘‘XtToolkitError’’.

To register a procedure to be called on nonfatal error conditions, use **XtAppSetWar-ningMsgHandler**.

XtErrorMsgHandler XtAppSetWarningMsgHandler(*app_context*, *msg_handler*)
  XtAppContext *app_context*;
  XtErrorMsgHandler *msg_handler*;

app_context Specifies the application context.

msg_handler Specifies the new nonfatal error procedure, which usually returns.

XtAppSetWarningMsgHandler returns a pointer to the previously installed high-level warning handler. The default high-level warning handler provided by the Intrinsics is named **_XtDe-faultWarningMsg** and 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 entries in *params*.

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**.

XtErrorHandler 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.

XtAppSetErrorHandler returns a pointer to the previously installed low-level fatal error handler. The default low-level error handler provided by the Intrinsics is **_XtDefaultError**. On POSIX-based systems, it prints the message to standard error and terminates the application. Fatal error message handlers should not return. If one does, subsequent Intrinsics 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 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**.

XtErrorHandler 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.

XtAppSetWarningHandler returns a pointer to the previously installed low-level warning handler. The default low-level warning handler provided by the Intrinsics is **_XtDefaultWarning**. On POSIX-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 to be reported.

Most programs should use **XtAppWarningMsg**, not **XtAppWarning**, to provide for customization and internationalization of warning messages.

11.10. Setting WM_COLORMAP_WINDOWS

A client may set the value of the WM_COLORMAP_WINDOWS property on a widget's window by calling **XtSetWMColormapWindows**.

void XtSetWMColormapWindows(*widget*, *list*, *count*)
    Widget *widget*;
    Widget* *list*;
    Cardinal *count*;

widget Specifies the widget on whose window the WM_COLORMAP_WINDOWS property will be stored. Must be of class Core or any subclass thereof.

list Specifies a list of widgets whose windows are potentially to be listed in the WM_COLORMAP_WINDOWS property.

count Specifies the number of widgets in *list*.

XtSetWMColormapWindows returns immediately if *widget* is not realized or if *count* is 0. Otherwise, **XtSetWMColormapWindows** constructs an ordered list of windows by examining each widget in *list* in turn and ignoring the widget if it is not realized, or adding the widget's window to the window list if the widget is realized and if its colormap resource is different from the colormap resources of all widgets whose windows are already on the window list.

Finally, **XtSetWMColormapWindows** stores the resulting window list in the WM_COLORMAP_WINDOWS property on the specified widget's window. Refer to Section 4.1.8 in the *Inter-Client Communication Conventions Manual* for details of the semantics of the WM_COLORMAP_WINDOWS property.

11.11. Finding File Names

The Intrinsics provide procedures to look for a file by name, allowing string substitutions in a list of file specifications. Two routines are provided for this: **XtFindFile** and **XtResolvePathname**. **XtFindFile** uses an arbitrary set of client-specified substitutions, and **XtResolvePathname** uses a set of standard substitutions corresponding to the *X/Open Portability Guide* language localization conventions. Most applications should use **XtResolvePathname**.

A string substitution is defined by a list of **Substitution** entries.

```
typedef struct {
        char match;
        String substitution;
} SubstitutionRec, *Substitution;
```

File name evaluation is handled in an operating-system-dependent fashion by an **XtFilePredicate** procedure.

typedef Boolean (*XtFilePredicate)(String);
    String *filename*;

filename Specifies a potential filename.

A file predicate procedure will be called with a string that is potentially a file name. It should return **True** if this string specifies a file that is appropriate for the intended use and **False** otherwise.

To search for a file using substitutions in a path list, use **XtFindFile**.

String XtFindFile(*path*, *substitutions*, *num_substitutions*, *predicate*)
    String *path*;
    Substitution *substitutions*;
    Cardinal *num_substitutions*;
    XtFilePredicate *predicate*;

path Specifies a path of file names, including substitution characters.

substitutions Specifies a list of substitutions to make into the path.

num_substitutions Specifies the number of substitutions passed in.

predicate Specifies a procedure called to judge each potential file name, or NULL.

The *path* parameter specifies a string that consists of a series of potential file names delimited by colons. Within each name, the percent character specifies a string substitution selected by the following character. The character sequence ‘‘%:’’ specifies an embedded colon that is not a delimiter; the sequence is replaced by a single colon. The character sequence ‘‘%%’’ specifies a percent character that does not introduce a substitution; the sequence is replaced by a single percent character. If a percent character is followed by any other character, **XtFindFile** looks through the specified *substitutions* for that character in the *match* field and if found replaces the percent and match characters with the string in the corresponding *substitution* field. A *substitution* field entry of NULL is equivalent to a pointer to an empty string. If the operating system does not interpret multiple embedded name separators in the path (i.e., ‘‘/’’ in POSIX) the same way as a single separator, **XtFindFile** will collapse multiple separators into a single one after performing all string substitutions. Except for collapsing embedded separators, the contents of the string substitutions are not interpreted by **XtFindFile** and may therefore contain any operating-system-dependent characters, including additional name separators. Each resulting string is passed to the predicate procedure until a string is found for which the procedure returns **True**; this string is the return value for **XtFindFile**. If no string yields a **True** return from the predicate, **XtFindFile** returns NULL.

If the *predicate* parameter is NULL, an internal procedure that checks if the file exists, is readable, and is not a directory will be used.

It is the responsibility of the caller to free the returned string using **XtFree** when it is no longer needed.

To search for a file using standard substitutions in a path list, use **XtResolvePathname**.

String XtResolvePathname(*display*, *type*, *filename*, *suffix*, *path*, *substitutions*, *num_substitutions*, *predicate*)
    Display **display*;
    String *type*, *filename*, *suffix*, *path*;
    Substitution *substitutions*;
    Cardinal *num_substitutions*;
    XtFilePredicate *predicate*;

display	Specifies the display to use to find the language for language substitutions.
type *filename* *suffix*	Specify values to substitute into the path.
path	Specifies the list of file specifications, or NULL.
substitutions	Specifies a list of additional substitutions to make into the path, or NULL.
num_substitutions	Specifies the number of entries in *substitutions*.
predicate	Specifies a procedure called to judge each potential file name, or NULL.

The substitutions specified by **XtResolvePathname** are determined from the value of the language string retrieved by **XtDisplayInitialize** for the specified display. To set the language for all applications specify ‘‘*xnlLanguage: *lang*’’ in the resource database. The format and content of the language string are implementation-defined. One suggested syntax is to compose the language string of three parts; a ‘‘language part’’, a ‘‘territory part’’ and a ‘‘codeset part’’. The manner in which this composition is accomplished is implementation-defined and the Intrinsics make no interpretation of the parts other than to use them in substitutions as described below.

XtResolvePathname calls **XtFindFile** with the following substitutions in addition to any passed by the caller and returns the value returned by **XtFindFile**:

%N The value of the *filename* parameter, or the application’s class name if *filename* is NULL.

%T The value of the *type* parameter.

%S The value of the *suffix* parameter.

%L The language string associated with the specified display.

%l The language part of the display’s language string.

%t The territory part of the display’s language string.

%c The codeset part of the display’s language string.

%C The customization string retrieved from the resource database associated with *dispay*.

If a path is passed to **XtResolvePathname**, it will be passed along to **XtFindFile**. If the *path* argument is NULL, the value of the **XFILESEARCHPATH** environment variable will be passed to **XtFindFile**. If **XFILESEARCHPATH** is not defined, an implementation-specific default path will be used which contains at least 6 entries. These entries must contain the following substitutions:

1.	%C, %N, %S, %T, %L	or	%C, %N, %S, %T, %l, %t, %c
2.	%C, %N, %S, %T, %l		
3.	%C, %N, %S, %T		
4.	%N, %S, %T, %L	or	%N, %S, %T, %l, %t, %c
5.	%N, %S, %T, %l		
6.	%N, %S, %T		

The order of these six entries within the path must be as given above. The order and use of substitutions within a given entry is implementation dependent. If the path begins with a colon, it will be preceded by %N%S. If the path includes two adjacent colons, **%N%S** will be inserted between them.

The *type* parameter is intended to be a category of files, usually being translated into a directory in the pathname. Possible values might include ‘‘app-defaults’’, ‘‘help’’, and ‘‘bitmap’’.

The *suffix* parameter is intended to be appended to the file name. Possible values might include ‘‘.txt’’, ‘‘.dat’’, and ‘‘.bm’’.

A suggested value for the default path on POSIX-based systems is

```
/usr/lib/X11/%L/%T/%N%C%S:/usr/lib/X11/%l/%T/%N%C%S:\
/usr/lib/X11/%T/%N%C%S:/usr/lib/X11/%L/%T/%N%S:\
/usr/lib/X11/%l/%T/%N%S:/usr/lib/X11/%T/%N%S
```

Using this example, if the user has specified a language, it will be used as a subdirectory of /usr/lib/X11 that will be searched for other files. If the desired file is not found there, the lookup will be tried again using just the language part of the specification. If the file is not there, it will be looked for in /usr/lib/X11. The *type* parameter is used as a subdirectory of the language directory or of /usr/lib/X11, and *suffix* is appended to the file name.

The customization string is obtained by querying the resource database currently associated with the display (the database returned by **XrmGetDatabase**) for the resource *application_name*.customization, class *application_class*.Customization where *application_name* and *application_class* are the values returned by **XtGetApplicationNameAndClass**. If no value is specified in the database, the empty string is used.

It is the responsibility of the caller to free the returned string using **XtFree** when it is no longer needed.

Chapter 12

Nonwidget Objects

Although widget writers are free to treat Core as the base class of the widget hierarchy, there are actually three classes above it. These classes are Object, RectObj, (Rectangle Object) and (*unnamed*) and members of these classes are referred to generically as *objects*. By convention, the term *widget* refers only to objects that are a subclass of Core, and the term *nonwidget* refers to objects that are not a subclass of Core. In the preceding portion of this specification, the interface descriptions indicate explicitly whether the generic *widget* argument is restricted to particular subclasses of Object. Sections 12.2.5, 12.3.5, and 12.5 summarize the permissible classes of the arguments to, and return values from, each of the Intrinsics routines.

12.1. Data Structures

In order not to conflict with previous widget code, the data structures used by nonwidget objects do not follow all the same conventions as those for widgets. In particular, the class records are not composed of parts but instead are complete data structures with filler for the widget fields they do not use. This allows the static class initializers for existing widgets to remain unchanged.

12.2. Object Objects

The Object object contains the definitions of fields common to all objects. It encapsulates the mechanisms for resource management. All objects and widgets are members of subclasses of Object, which is defined by the **ObjectClassPart** and **ObjectPart** structures.

12.2.1. ObjectClassPart Structure

The common fields for all object classes are defined in the **ObjectClassPart** structure. All fields have the same purpose, function, and restrictions as the corresponding fields in **CoreClassPart**; fields whose names are obj*n* for some integer *n* are not used for Object, but exist to pad the data structure so that it matches Core's class record. The class record initialization must fill all obj*n* fields with NULL or zero as appropriate to the type.

```
typedef struct _ObjectClassPart {
        WidgetClass superclass;
        String class_name;
        Cardinal widget_size;
        XtProc class_initialize;
        XtWidgetClassProc class_part_initialize;
        XtEnum class_inited;
        XtInitProc initialize;
        XtArgsProc initialize_hook;
        XtProc obj1;
        XtPointer obj2;
        Cardinal obj3;
        XtResourceList resources;
        Cardinal num_resources;
        XrmClass xrm_class;
```

```
    Boolean obj4;
        XtEnum obj5;
        Boolean obj6;
        Boolean obj7;
        XtWidgetProc destroy;
        XtProc obj8;
        XtProc obj9;
        XtSetValuesFunc set_values;
        XtArgsFunc set_values_hook;
        XtProc obj10;
        XtArgsProc get_values_hook;
        XtProc obj11;
        XtVersionType version;
        XtPointer callback_private;
        String obj12;
        XtProc obj13;
        XtProc obj14;
        XtPointer extension;
} ObjectClassPart;
```

The prototypical **ObjectClass** consists of just the **ObjectClassPart**.

```
typedef struct _ObjectClassRec {
        ObjectClassPart object_class;
} ObjectClassRec, *ObjectClass;
```

The predefined class record and pointer for **ObjectClassRec** are

In **IntrinsicP.h**:
extern ObjectClassRec objectClassRec;

In **Intrinsic.h**:
extern WidgetClass objectClass;

The opaque types **Object** and **ObjectClass** and the opaque variable **objectClass** are defined for generic actions on objects. **Intrinsic.h** uses an incomplete structure definition to ensure that the compiler catches attempts to access private data:

typedef struct _ObjectClassRec* ObjectClass;

12.2.2. ObjectPart Structure

The common fields for all object instances are defined in the **ObjectPart** structure. All fields have the same meaning as the corresponding fields in **CorePart**.

```
typedef struct _ObjectPart {
        Widget self;
        WidgetClass widget_class;
        Widget parent;
        Boolean being_destroyed;
        XtCallbackList destroy_callbacks;
        XtPointer constraints;
} ObjectPart;
```

All object instances have the Object fields as their first component. The prototypical type **Object** is defined with only this set of fields. Various routines can cast object pointers, as needed, to specific object types.

In **IntrinsicP.h**:
```
typedef struct _ObjectRec {
        ObjectPart object;
} ObjectRec, *Object;
```

In **Intrinsic.h**:
```
typedef struct _ObjectRec *Object;
```

12.2.3. Object Resources

The resource names, classes, and representation types specified in the **objectClassRec** resource list are

Name	Class	Representation
XtNdestroyCallback	XtCCallback	XtRCallback

12.2.4. ObjectPart Default Values

All fields in **ObjectPart** have the same default values as the corresponding fields in **CorePart**.

12.2.5. Object Arguments To Intrinsics Routines

The WidgetClass arguments to the following procedures may be **objectClass** or any subclass:

> **XtInitializeWidgetClass**, **XtCreateWidget**, **XtVaCreateWidget**
>
> **XtIsSubclass**, **XtCheckSubclass**
>
> **XtGetResourceList**, **XtGetConstraintResourceList**

The Widget arguments to the following procedures may be of class Object or any subclass:

> **XtCreateWidget**, **XtVaCreateWidget**
>
> **XtAddCallback**, **XtAddCallbacks**, **XtRemoveCallback**, **XtRemoveCallbacks**, **XtRemoveAllCallbacks**, **XtCallCallbacks**, **XtHasCallbacks**, **XtCallCallbackList**
>
> **XtClass**, **XtSuperclass**, **XtIsSubclass**, **XtCheckSubclass**, **XtIsObject**, **XtIsRectObj**, **XtIsWidget**, **XtIsComposite**, **XtIsConstraint**, **XtIsShell**, **XtIsOverrideShell**, **XtIsWMShell**, **XtIsVendorShell**, **XtIsTransientShell**, **XtIsToplevelShell**, **XtIsApplicationShell**.
>
> **XtIsManaged**, **XtIsSensitive**
> (both will return **False** if argument is not a subclass of RectObj)
>
> **XtIsRealized**
> (returns the state of the nearest windowed ancestor if class of argument is not a subclass of Core)

XtWidgetToApplicationContext
XtDestroyWidget
XtParent, **XtDisplayOfObject**, **XtScreenOfObject**, **XtWindowOfObject**
XtSetKeyboardFocus (descendant)
XtGetGC, **XtReleaseGC**
XtName
XtSetValues, **XtGetValues**, **XtVaSetValues**, **XtVaGetValues**
XtGetSubresources, **XtGetApplicationResources**, **XtVaGetSubresources**, **XtVaGetApplicationResources**
XtConvert, **XtConvertAndStore**

The return value of the following procedures will be of class Object or a subclass:

XtCreateWidget, **XtVaCreateWidget**
XtParent
XtNameToWidget

The return value of the following procedures will be **objectClass** or a subclass:

XtClass, **XtSuperclass**

12.2.6. Use of Objects

The Object class exists to enable programmers to use the Intrinsics' classing and resource-handling mechanisms for things smaller and simpler than widgets. Objects make obsolete many common uses of subresources as described in sections 9.4, 9.7.2.4, and 9.7.2.5.

Composite widget classes that wish to accept nonwidget children must set the *accepts_objects* field in the **CompositeClassExtension** structure to **True**. **XtCreateWidget** will otherwise generate an error message on an attempt to create a nonwidget child.

Of the classes defined by the Intrinsics, only ApplicationShell accepts nonwidget children, and the class of any nonwidget child must not be **rectObjClass** or any subclass. The intent of allowing Object children of ApplicationShell is to provide clients a simple mechanism for establishing the resource-naming root of an object hierarchy.

12.3. Rectangle Objects

The class of rectangle objects is a subclass of Object that represents rectangular areas. It encapsulates the mechanisms for geometry management, and is called RectObj to avoid conflict with the Xlib **Rectangle** data type.

12.3.1. RectObjClassPart Structure

As with the **ObjectClassPart** structure, all fields in the **RectObjClassPart** structure have the same purpose and function as the corresponding fields in **CoreClassPart**; fields whose names are rect*n* for some integer *n* are not used for RectObj but exist to pad the data structure so that it matches Core's class record. The class record initialization must fill all rect*n* fields with NULL or zero as appropriate to the type.

```
typedef struct _RectObjClassPart {
        WidgetClass superclass;
        String class_name;
        Cardinal widget_size;
        XtProc class_initialize;
        XtWidgetClassProc class_part_initialize;
        XtEnum class_inited;
        XtInitProc initialize;
        XtArgsProc initialize_hook;
        XtProc rect1;
        XtPointer rect2;
        Cardinal rect3;
        XtResourceList resources;
        Cardinal num_resources;
        XrmClass xrm_class;
        Boolean rect4;
        XtEnum rect5;
        Boolean rect6;
        Boolean rect7;
        XtWidgetProc destroy;
        XtWidgetProc resize;
        XtExposeProc expose;
        XtSetValuesFunc set_values;
        XtArgsFunc set_values_hook;
        XtAlmostProc set_values_almost;
        XtArgsProc get_values_hook;
        XtProc rect9;
        XtVersionType version;
        XtPointer callback_private;
        String rect10;
        XtGeometryHandler query_geometry;
        XtProc rect11;
        XtPointer extension ;
} RectObjClassPart;
```

The RectObj class record consists of just the **RectObjClassPart**.

```
typedef struct _RectObjClassRec {
        RectObjClassPart rect_class;
} RectObjClassRec, *RectObjClass;
```

The predefined class record and pointer for **RectObjClassRec** are

In **Intrinsic.h**:

```
extern RectObjClassRec rectObjClassRec;
```

In **Intrinsic.h**:

```
extern WidgetClass rectObjClass;
```

The opaque types **RectObj** and **RectObjClass** and the opaque variable **rectObjClass** are defined for generic actions on objects whose class is RectObj or a subclass of RectObj. **Intrinsic.h** uses an incomplete structure definition to ensure that the compiler catches attempts to access private data:

```
typedef struct _RectObjClassRec* RectObjClass;
```

12.3.2. RectObjPart Structure

In addition to the **ObjectPart** fields, RectObj objects have the following fields defined in the **RectObjPart** structure. All fields have the same meaning as the corresponding field in **CorePart**.

```
typedef struct _RectObjPart {
        Position x, y;
        Dimension width, height;
        Dimension border_width;
        Boolean managed;
        Boolean sensitive;
        Boolean ancestor_sensitive;
} RectObjPart;
```

RectObj objects have the RectObj fields immediately following the Object fields.

```
typedef struct _RectObjRec {
        ObjectPart object;
        RectObjPart rectangle;
} RectObjRec, *RectObj;
```

In **Intrinsic.h**:

```
typedef struct _RectObjRec* RectObj;
```

12.3.3. RectObj Resources

The resource names, classes, and representation types that are specified in the **rectObjClassRec** resource list are

Name	Class	Representation
XtNancestorSensitive	XtCSensitive	XtRBoolean
XtNborderWidth	XtCBorderWidth	XtRDimension
XtNheight	XtCHeight	XtRDimension
XtNsensitive	XtCSensitive	XtRBoolean
XtNwidth	XtCWidth	XtRDimension
XtNx	XtCPosition	XtRPosition
XtNy	XtCPosition	XtRPosition

12.3.4. RectObjPart Default Values

All fields in **RectObjPart** have the same default values as the corresponding fields in **CorePart**.

12.3.5. Widget Arguments To Intrinsics Routines

The WidgetClass arguments to the following procedures may be **rectObjClass** or any subclass:

XtCreateManagedWidget, **XtVaCreateManagedWidget**

The Widget arguments to the following procedures may be of class RectObj or any subclass:

XtConfigureWidget, **XtMoveWidget**, **XtResizeWidget**
XtMakeGeometryRequest, **XtMakeResizeRequest**
XtManageChildren, **XtManageChild**, **XtUnmanageChildren**, **XtUnmanageChild**
XtQueryGeometry
XtSetSensitive
XtTranslateCoords

The return value of the following procedures will be of class RectObj or a subclass:

XtCreateManagedWidget, **XtVaCreateManagedWidget**

12.3.6. Use of Rectangle Objects

RectObj can be subclassed to provide widgetlike objects (sometimes called gadgets) that do not use windows and do not have features often unused in simple widgets. This can save memory resources both in the server and in applications but requires additional support code in the parent. In the following discussion, *rectobj* refers only to objects whose class is RectObj or a subclass of RectObj but not Core or a subclass of Core.

Composite widget classes that wish to accept rectobj children must set the *accepts_objects* field in the **CompositeClassExtension** extension structure to **True**. **XtCreateWidget** or **XtCreateManagedWidget** will otherwise generate an error if called to create a nonwidget child. If the composite widget supports only children of class RectObj or a subclass (i.e., not of the general Object class), it must declare an insert_child procedure and check the subclass of each new child in that procedure. None of the classes defined by the Intrinsics accept rectobj children.

If gadgets are defined in an object set, the parent is responsible for much more than the parent of a widget. The parent must request and handle input events that occur for the gadget and is responsible for making sure that when it receives an exposure event the gadget children get drawn correctly. Rectobj children may have expose procedures specified in their class records, but the parent is free to ignore them, instead drawing the contents of the child itself. This can potentially save graphics context switching. The precise contents of the exposure event and region arguments to the RectObj expose procedure are not specified by the Intrinsics; a particular rectangle object is free to define the coordinate system origin (self-relative or parent-relative) and whether or not the rectangle or region is assumed to have been intersected with the visible region of the object.

In general, it is expected that a composite widget that accepts nonwidget children will document those children it is able to handle, since a gadget cannot be viewed as a completely self-contained entity, as can a widget. Since a particular composite widget class is usually designed to handle nonwidget children of only a limited set of classes, it should check the classes of newly added children in its insert_child procedure to make sure that it can deal with them.

The Intrinsics will clear areas of a parent window obscured by rectobj children, causing exposure events, under the following circumstances:

- A rectobj child is managed or unmanaged.
- In a call to **XtSetValues** on a rectobj child, one or more of the set_values procedures returns **True**.
- In a call to **XtConfigureWidget** on a rectobj child, areas will be cleared corresponding to both the old and the new child geometries, including the border, if the geometry changes.
- In a call to **XtMoveWidget** on a rectobj child, areas will be cleared corresponding to both the old and the new child geometries, including the border, if the geometry changes.

- In a call to **XtResizeWidget** on a rectobj child, an single rectangle will be cleared corresponding to the larger of the old and the new child geometries if they are different.
- In a call to **XtMakeGeometryRequest** (or **XtMakeResizeRequest**) on a rectobj child with **XtQueryOnly** not set, if the manager returns **XtGeometryYes**, two rectangles will be cleared corresponding to both the old and the new child geometries.

Stacking order is not supported for rectobj children. Composite widgets with rectobj children are free to define any semantics desired if the child geometries overlap, including making this an error.

When a rectobj is playing the role of a widget, developers must be reminded to avoid making assumptions about the object passed in the Widget argument to a callback procedure.

12.4. Undeclared Class

The Intrinsics define an unnamed class between RectObj and Core for possible future use by the X Consortium. The only assumptions that may be made about the unnamed class are

- the *core_class.superclass* field of **coreWidgetClassRec** contains a pointer to the unnamed class record.
- a pointer to the unnamed class record when dereferenced as an **ObjectClass** will contain a pointer to **rectObjClassRec** in its *object_class.superclass* field.

Except for the above, the contents of the class record for this class and the result of an attempt to subclass or to create a widget of this unnamed class are undefined.

12.5. Widget Arguments To Intrinsics Routines

The WidgetClass arguments to the following procedures must be of class Shell or a subclass:

> **XtCreatePopupShell**, **XtVaCreatePopupShell**, **XtAppCreateShell**, **XtVaAppCreateShell**

The Widget arguments to the following procedures must be of class Core or any subclass:

> **XtCreatePopupShell**, **XtVaCreatePopupShell**
>
> **XtAddEventHandler**, **XtAddRawEventHandler**, **XtRemoveEventHandler**, **XtRemoveRawEventHandler**, **XtInsertEventHandler**, **XtInsertRawEventHandler**
>
> **XtAddGrab**, **XtRemoveGrab**, **XtGrabKey**, **XtGrabKeyboard**, **XtUngrabKey**, **XtUngrabKeyboard**, **XtGrabButton**, **XtGrabPointer**, **XtUngrabButton**, **XtUngrabPointer**
>
> **XtBuildEventMask**
>
> **XtCreateWindow**, **XtDisplay**, **XtScreen**, **XtWindow**
>
> **XtNameToWidget**
>
> **XtGetSelectionValue**, **XtGetSelectionValues**, **XtOwnSelection**, **XtDisownSelection**, **XtOwnSelectionIncremental**, **XtGetSelectionValueIncremental**, **XtGetSelectionValuesIncremental**,
> **XtGetSelectionRequest**
>
> **XtInstallAccelerators**, **XtInstallAllAccelerators** (both destination and source)
>
> **XtAugmentTranslations**, **XtOverrideTranslations**, **XtUninstallTranslations**, **XtCallActionProc**

XtMapWidget, **XtUnmapWidget**
XtRealizeWidget, **XtUnrealizeWidget**
XtSetMappedWhenManaged
XtCallAcceptFocus, **XtSetKeyboardFocus** (subtree)
XtResizeWindow
XtSetWMColormapWindows

The Widget arguments to the following procedures must be of class Composite or any subclass:

XtCreateManagedWidget, **XtVaCreateManagedWidget**

The Widget arguments to the following procedures must be of a subclass of Shell:

XtPopdown, **XtCallbackPopdown**, **XtPopup**, **XtCallbackNone**, **XtCallbackNonexclusive**, **XtCallbackExclusive**, **XtPopupSpringLoaded**

The return value of the following procedure will be of class Core or a subclass:

XtWindowToWidget

The return value of the following procedures will be of a subclass of Shell :

XtAppCreateShell, **XtVaAppCreateShell**, **XtAppInitialize**, **XtVaAppInitialize**, **XtCreatePopupShell**, **XtVaCreatePopupShell**

Chapter 13

Evolution of The Intrinsics

The interfaces described by this specification have undergone several sets of revisions in the course of adoption as an X Consortium standard specification. Having now been adopted by the Consortium as a standard part of the X Window System, it is expected that this and future revisions will retain backward compatibility in the sense that fully conforming implementations of these specifications may be produced that provide source compatibility with widgets and applications written to previous Consortium standard revisions.

The Intrinsics do not place any special requirement on widget programmers to retain source or binary compatibility for their widgets as they evolve, but several conventions have been established to assist those developers who want to provide such compatibility.

In particular, widget programmers may wish to conform to the convention described in Section 1.6.12 when defining class extension records.

13.1. Determining Specification Revision Level

Widget and application developers who wish to maintain a common source pool that will build properly with implementations of the Intrinsics at different revision levels of these specifications but that take advantage of newer features added in later revisions may use the symbolic macro **XtSpecificationRelease**.

```
#define XtSpecificationRelease 5
```

As the symbol **XtSpecificationRelease** was new to Release 4, widgets and applications desiring to build against earlier implementations should test for the presence of this symbol and assume only Release 3 interfaces if the definition is not present.

13.2. Release 3 to Release 4 Compatibility

At the data structure level, Release 4 retains binary compatibility with Release 3 (the first X Consortium standard release) for all data structures except **WMShellPart, TopLevelShellPart**, and **TransientShellPart**. Release 4 changed the argument type to most procedures that now take arguments of type **XtPointer** and structure members that are now of type **XtPointer** in order to avoid potential ANSI C conformance problems. It is expected that most implementations will be binary compatible with the previous definition.

Two fields in **CoreClassPart** were changed from **Boolean** to **XtEnum** to allow implementations additional freedom in specifying the representations of each. This change should require no source modification.

13.2.1. Additional Arguments

Arguments were added to the procedure definitions for **XtInitProc**, **XtSetValuesFunc**, and **XtEventHandler** to provide more information and to allow event handlers to abort further dispatching of the current event (caution is advised!). The added arguments to **XtInitProc** and **XtSetValuesFunc** make the initialize_hook and set_values_hook methods obsolete, but the hooks have been retained for those widgets that used them in Release 3.

13.2.2. set_values_almost Procedures

The use of the arguments by a set_values_almost procedure was poorly described in Release 3 and was inconsistent with other conventions.

The current specification for the manner in which a set_values_almost procedure returns information to the Intrinsics is not compatible with the Release 3 specification, and all widget implementations should verify that any set_values_almost procedures conform to the current interface.

No known implementation of the Intrinsics correctly implemented the Release 3 interface, so it is expected that the impact of this specification change is small.

13.2.3. Query Geometry

A composite widget layout routine that calls **XtQueryGeometry** is now expected to store the complete new geometry in the intended structure; previously the specification said ‘‘store the changes it intends to make’’. Only by storing the complete geometry does the child have any way to know what other parts of the geometry may still be flexible. Existing widgets should not be affected by this, except to take advantage of the new information.

13.2.4. unrealizeCallback Callback List

In order to provide a mechanism for widgets to be notified when they become unrealized through a call to **XtUnrealizeWidget**, the callback list name ‘‘unrealizeCallback’’ has been defined by the Intrinsics. A widget class that requires notification on unrealize may declare a callback list resource by this name. No class is required to declare this resource, but any class that did so in a prior revision may find it necessary to modify the resource name if it does not wish to use the new semantics.

13.2.5. Subclasses of WMShell

The formal adoption of the *Inter-Client Communication Conventions Manual* as an X Consortium standard has meant the addition of four fields to **WMShellPart** and one field to **TopLevelShellPart**. In deference to some widget libraries that had developed their own additional conventions to provide binary compatibility, these five new fields were added at the end of the respective data structures.

To provide more convenience for TransientShells, a field was added to the previously empty **TransientShellPart**. On some architectures the size of the part structure will not have changed as a result of this.

Any widget implementation whose class is a subclass of TopLevelShell or TransientShell must at minimum be recompiled with the new data structure declarations. Because **WMShellPart** no longer contains a contiguous **XSizeHints** data structure, a subclass that expected to do a single structure assignment of an **XSizeHints** structure to the *size_hints* field of **WMShellPart** must be revised, though the old fields remain at the same positions within **WMShellPart**.

13.2.6. Resource Type Converters

A new interface declaration for resource type converters was defined to provide more information to converters, to support conversion cache cleanup with resource reference counting, and to allow additional procedures to be declared to free resources. The old interfaces remain (in the compatibility section) and a new set of procedures was defined that work only with the new type converter interface.

In the now obsolete old type converter interface, converters are reminded that they must return the size of the converted value as well as its address. The example indicated this, but the description of **XtConverter** was incomplete.

13.2.7. KeySym Case Conversion Procedure

The specification for the **XtCaseProc** function type has been changed to match the Release 3 implementation, which included necessary additional information required by the function (a pointer to the display connection), and corrects the argument type of the source KeySym parameter. No known implementation of the Intrinsics implemented the previously documented interface.

13.2.8. Nonwidget Objects

Formal support for nonwidget objects is new to Release 4. A prototype implementation was latent in at least one Release 3 implementation of the Intrinsics, but the specification has changed somewhat. The most significant change is the requirement for a composite widget to declare the **CompositeClassExtension** record with the *accepts_objects* field set to **True** in order to permit a client to create a nonwidget child.

The addition of this extension field ensures that composite widgets written under Release 3 will not encounter unexpected errors if an application attempts to create a nonwidget child. In Release 4 there is no requirement that all composite widgets implement the extra functionality required to manage windowless children, so the *accept_objects* field allows a composite widget to declare that it is not prepared to do so.

13.3. Release 4 to Release 5 Compatibility

At the data structure level, Release 5 retains complete binary compatibility with release 4. The specification of the **ObjectPart**, **RectObjPart**, **CorePart**, **CompositePart**, **ShellPart**, **WMShellPart**, **TopLevelShellPart**, and **ApplicationShellPart** instance records was made less strict to permit implementations to add internal fields to these structures. Any implementation that chooses to do so would, of course, force a recompilation. The Xlib specification for **XrmValue** and **XrmOptionDescRec** was updated to use a new type, **XPointer**, for the *addr* and *value* fields respectively, to avoid ANSI C conformance problems. The definition of **XPointer** is binary compatible with the previous implementation.

13.3.1. baseTranslations Resource

A new pseudo-resource, XtNbaseTranslations, was defined to permit application developers to specify translation tables in application defaults files while still giving end users the ability to augment or override individual event sequences. This change will affect only those applications that wish to take advantage of the new functionality, or those widgets that may have previously defined a resource named ‘‘baseTranslations’’.

Applications wishing to take advantage of the new functionality would change their application defaults file, e.g., from

app.widget.translations: *value*

to

app.widget.baseTranslations: *value*

If it is important to the application to preserve complete compatibility of the defaults file between different versions of the application running under Release 4 and Release 5, the full translations can be replicated in both the ‘‘translations’’ and the ‘‘baseTranslations’’ resource.

13.3.2. Resource File Search Path

The current specification allows implementations greater flexibility in defining the directory structure used to hold the application class and per-user application defaults files. Previous specifications required the substitution strings to appear in the default path in a certain order, preventing sites from collecting all the files for a specific application together in one directory. The Release 5 specification allows the default path to specify the substitution strings in any order within a single path entry. Users will need to pay close attention to the documentation for the specific implementation to know where to find these files and how to specify their own **XFILESEARCHPATH** and **XUSERFILESEARCHPATH** values when overriding the system defaults.

13.3.3. Customization Resource

XtResolvePathname supports a new substitution string, %C, for specifying separate application class resource files according to arbitrary user-specified categories. The primary motivation for this addition was separate monochrome and color application class defaults files. The substitution value is obtained by querying the current resource database for the application resource name ‘‘customization’’, class ‘‘Customization’’. Any application that previously used this resource name and class will need to be aware of the possibly conflicting semantics.

13.3.4. Per-Screen Resource Database

To allow a user to specify separate preferences for each screen of a display, a per-screen resource specification string has been added and multiple resource databases are created; one for each screen. This will affect any application that modified the (formerly unique) resource database associated with the display subsequent to the Intrinsics database initialization. Such applications will need to be aware of the particular screen on which each shell widget is to be created.

Although the wording of the specification changed substantially in the description of the process by which the resource database(s) is initialized, the net effect is the same as in prior releases with the exception of the added per-screen resource specification and the new customization substitution string in **XtResolvePathname**.

13.3.5. Internationalization of Applications

Internationalization as defined by ANSI is a technology that allows support of an application in a single locale. In adding support for internationalization to the Intrinsics the restrictions of this model have been followed. In particular, the new Intrinsics interfaces are designed to not preclude an application from using other alternatives. For this reason, no Intrinsics routine makes a call to establish the locale. However, a convenience routine to establish the locale at initialize time has been provided, in the form of a default procedure that must be explicitly installed if the application desires ANSI C locale behavior.

As many objects in X, particularly resource databases, now inherit the global locale when they are created, applications wishing to use the ANSI C locale model should use the new function **XtSetLanguageProc** to do so.

The internationalization additions also define event filters as a part of the Xlib Input Method specifications. The Intrinsics enable the use of event filters through additions to **XtDispatchEvent**. Applications that may not be dispatching all events through **XtDispatchEvent** should be reviewed in the context of this new input method mechanism.

In order to permit internationalization of error messages the name and path of the error database file is now allowed to be implementation dependent. No adequate standard mechanism has yet been suggested to allow the Intrinsics to locate the database from localization information supplied by the client.

The previous specification for the syntax of the language string specified by **xnlLanguage** has been dropped to avoid potential conflicts with other standards. The language string syntax is now implementation-defined. The example syntax cited is consistent with the previous specification.

13.3.6. Permanently Allocated Strings

In order to permit additional memory savings, an Xlib interface was added to allow the resource manager to avoid copying certain string constants. The Intrinsics specification was updated to explicitly require the Object *class_name*, *resource_name*, *resource_class*, *resource_type*, *default_type* in resource tables, Core *actions string* field, and Constraint *resource_name*, *resource_class*, *resource_type*, and *default_type* resource fields to be permanently allocated. This explicit requirement is expected to affect only applications that may create and destroy classes on the fly.

13.3.7. Arguments to Existing Functions

The *args* argument to **XtAppInitialize**, **XtVaAppInitialize**, **XtOpenDisplay**, **XtDisplayInitialize**, and **XtInitialize** were changed from **Cardinal*** to int* to conform to pre-existing convention and avoid otherwise annoying typecasting in ANSI C environments.

Appendix A

Resource File Format

A resource file contains text representing the default resource values for an application or set of applications.

The format of resource files is defined by *Xlib – C Language X Interface* and is reproduced here for convenience only.

The format of a resource specification is

```
ResourceLine     = Comment | IncludeFile | ResourceSpec | <empty line>
Comment          = ‘‘!’’ {<any character except null or newline>}
IncludeFile      = ‘‘#’’ WhiteSpace ‘‘include’’ WhiteSpace FileName WhiteSpace
FileName         = <valid filename for operating system>
ResourceSpec     = WhiteSpace ResourceName WhiteSpace ‘‘:’’ WhiteSpace Value
ResourceName     = [Binding] {Component Binding} ComponentName
Binding          = ‘‘.’’ | ‘‘*’’
WhiteSpace       = {<space> | <horizontal tab>}
Component        = ‘‘?’’ | ComponentName
ComponentName    = NameChar {NameChar}
NameChar         = ‘‘a’’-‘‘z’’ | ‘‘A’’-‘‘Z’’ | ‘‘0’’-‘‘9’’ | ‘‘_’’ | ‘‘-’’
Value            = {<any character except null or unescaped newline>}
```

Elements separated by vertical bar (|) are alternatives. Curly braces ({...}) indicate zero or more repetitions of the enclosed elements. Square brackets ([...]) indicate that the enclosed element is optional. Quotes (‘‘...’’) are used around literal characters.

If the last character on a line is a backslash (\), that line is assumed to continue on the next line.

To allow a Value to begin with whitespace, the two-character sequence ‘‘*space*’’ (backslash followed by space) is recognized and replaced by a space character, and the two-character sequence ‘‘*tab*’’ (backslash followed by horizontal tab) is recognized and replaced by a horizontal tab character.

To allow a Value to contain embedded newline characters, the two-character sequence ‘‘\n’’ is recognized and replaced by a newline character. To allow a Value to be broken across multiple lines in a text file, the two-character sequence ‘‘*newline*’’ (backslash followed by newline) is recognized and removed from the value.

To allow a Value to contain arbitrary character codes, the four-character sequence ‘‘*nnn*’’, where each *n* is a digit character in the range of ‘‘0’’–‘‘7’’, is recognized and replaced with a single byte that contains the octal value specified by the sequence. Finally, the two-character sequence ‘‘\\’’ is recognized and replaced with a single backslash.

Appendix B

Translation Table Syntax

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’’
( a | b )  Means either ‘‘a’’ or ‘‘b’’
\n         Is the newline character
```

All terminals are enclosed in double quotation marks (‘‘ ’’). Informal descriptions are enclosed in angle brackets (< >).

Syntax

The syntax of a translation table 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> | escape_char} [‘‘\\’’ ] ‘‘"’’
escape_char       = ‘‘\"’’
unquoted_string   = {<Latin 1 character except space, tab, ‘‘,’’, ‘‘\n’’, ‘‘)’’>}
```

The *params* field is parsed into a list of **String** values that will be passed to the named action procedure. A *quoted string* may contain an embedded quotation mark if the quotation mark is preceded by a single backslash (\). The three-character sequence ‘‘\\"’’ is interpreted as ‘‘single backslash followed by end-of-string’’.

Modifier Names

The modifier field is used to specify standard 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 *X Window System Protocol*, Section 5. 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 both an exclamation point (!) and a colon (:) are specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and that no other modifiers except the standard modifiers can be asserted. Any standard modifiers in the event are applied as for colon (:) above.
- 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 (ˆ) 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 (ˆ), 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 an exclamation point with no modifers.

Modifier	Abbreviation	Meaning
Ctrl	c	Control modifier bit
Shift	s	Shift modifier bit
Lock	l	Lock modifier bit
Meta	m	Meta key modifier
Hyper	h	Hyper key modifier
Super	su	Super key modifier
Alt	a	Alt key modifier
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

Modifier	Abbreviation	Meaning
None		No modifiers
Any		Any modifier combination

A key modifier is any modifier bit one of whose corresponding KeyCodes 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 key modifiers is extensible. The ‘‘@’’ <keysym> syntax means any modifier bit whose corresponding KeyCode contains the specified KeySym name.

A modifier_list/KeySym combination in a translation matches a modifiers/KeyCode combination in an event in the following ways:

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 event-type field describes XEvent types. In addition to the standard Xlib symbolic event type names, the following event type synonyms are defined:

Type	Meaning
Key	**KeyPress**
KeyDown	**KeyPress**
KeyUp	**KeyRelease**
BtnDown	**ButtonPress**
BtnUp	**ButtonRelease**
Motion	**MotionNotify**
PtrMoved	**MotionNotify**
MouseMoved	**MotionNotify**
Enter	**EnterNotify**
EnterWindow	**EnterNotify**
Leave	**LeaveNotify**
LeaveWindow	**LeaveNotify**
FocusIn	**FocusIn**
FocusOut	**FocusOut**
Keymap	**KeymapNotify**
Expose	**Expose**
GrExp	**GraphicsExpose**
NoExp	**NoExpose**
Visible	**VisibilityNotify**
Create	**CreateNotify**

Type	Meaning
Destroy	**DestroyNotify**
Unmap	**UnmapNotify**
Map	**MapNotify**
MapReq	**MapRequest**
Reparent	**ReparentNotify**
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	Event Type	Including
Ctrl	**KeyPress**	with Control modifier
Meta	**KeyPress**	with Meta modifier
Shift	**KeyPress**	with Shift modifier
Btn1Down	**ButtonPress**	with Button1 detail
Btn1Up	**ButtonRelease**	with Button1 detail
Btn2Down	**ButtonPress**	with Button2 detail
Btn2Up	**ButtonRelease**	with Button2 detail
Btn3Down	**ButtonPress**	with Button3 detail
Btn3Up	**ButtonRelease**	with Button3 detail
Btn4Down	**ButtonPress**	with Button4 detail
Btn4Up	**ButtonRelease**	with Button4 detail
Btn5Down	**ButtonPress**	with Button5 detail
Btn5Up	**ButtonRelease**	with Button5 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 the corresponding event as described by *X Window System Protocol*, Section 11. The detail field is supported for the following event types:

Event	Event Field
KeyPress	KeySym from event *detail* (keycode)

KeyRelease	KeySym from event *detail* (keycode)
ButtonPress	button from event *detail*
ButtonRelease	button from event *detail*
MotionNotify	event *detail*
EnterNotify	event *mode*
LeaveNotify	event *mode*
FocusIn	event *mode*
FocusOut	event *mode*
PropertyNotify	*atom*
SelectionClear	*selection*
SelectionRequest	*selection*
SelectionNotify	*selection*
ClientMessage	*type*
MappingNotify	*request*

If the event type is **KeyPress** or **KeyRelease**, the detail field specifies a KeySym name in standard format which is matched against the event as described above, for example, <Key>A.

For the **PropertyNotify**, **SelectionClear**, **SelectionRequest**, **SelectionNotify** and **ClientMessage** events the detail field is specified as an atom name; for example, <Message>WM_PROTOCOLS. For the **MotionNotify**, **EnterNotify**, **LeaveNotify**, **FocusIn**, **FocusOut** and **MappingNotify** events, either the symbolic constants as defined by *X Window System Protocol*, Section 11, or the numeric values may be specified.

If no detail field is specified, then any value in the event detail is accepted as a match.

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** procedure to describe the accelerators installed on that widget. The canonical representation of a translation table 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> | escape_char} [‘‘\\’’ ] ‘‘"’’
escape_char        = ‘‘\"’’
```

The canonical modifier names are

Ctrl	Mod1	Button1
Shift	Mod2	Button2
Lock	Mod3	Button3
	Mod4	Button4
	Mod5	Button5

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 Button1 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 Button1 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 noninitial 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 Button1 Up with Shift and Meta, use this specification:

  ```
  Shift Meta <Btn1Down>, Shift Meta<Btn1Up>: gyre()
  ```

- For multiple clicks greater or equal to a minimum number, a plus sign (+) may be appended to the final (rightmost) count in an event sequence. The actions will be invoked on the *count*-th click and each subsequent one arriving within the multi-click time interval. 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 Button1 Down and Button2 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 (!).

Appendix C

Compatibility Functions

In prototype versions of the X 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 the Intrinsics, a single generic **XtCreateWidget** performs most of the common work and then calls the initialize procedure implemented for the particular widget class.

Each Composite class also implemented the procedures Xt<*Widget*>Add and an Xt<*Widget*>Delete (for example, **XtButtonBoxAddButton** and **XtButtonBoxDeleteButton**). In the Intrinsics, 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 to the change_managed procedure is delayed until realization time.

Old style calls can be implemented in the X Toolkit by defining one-line procedures or macros that invoke a generic routine. For example, you could define the macro **XtLabelCreate** as:

```
#define XtLabelCreate(name, parent, args, num_args) \
    ((LabelWidget) XtCreateWidget(name, labelWidgetClass,   parent, args, num_args))
```

Pop-up shells in some of the prototypes automatically performed 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 without application contexts, the following routines exist: **XtInitialize**, **XtMainLoop**, **XtNextEvent**, **XtProcessEvent**, **XtPeekEvent**, **XtPending**, **XtAddInput**, **XtAddTimeOut**, **XtAddWorkProc**, **XtCreateApplicationShell**, **XtAddActions**, **XtSetSelectionTimeout**, and **XtGetSelectionTimeout**.

```
Widget XtInitialize(shell_name, application_class, options, num_options, argc, argv)
    String shell_name;
    String application_class;
    XrmOptionDescRec options[];
    Cardinal num_options;
    int *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**.

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.

XtInitialize calls **XtToolkitInitialize** to initialize the toolkit internals, creates a default application context for use by the other convenience routines, calls **XtOpenDisplay** with *display_string* NULL and *application_name* NULL, and finally calls **XtAppCreateShell** with *application_name* NULL and returns the created shell. The semantics of calling **XtInitialize** more than once are undefined. This routine has been replaced by **XtAppInitialize**.

void XtMainLoop(void)

XtMainLoop first reads the next alternate input, timer, or X event by calling **XtNextEvent**. Then it dispatches this to the appropriate registered procedure by calling **XtDispatchEvent**. This routine has been replaced by **XtAppMainLoop**.

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 alternate 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 X event, timeout, or alternate input source (depending on the value of *mask*), blocking 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 nonzero 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 nonzero 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()

XtPending returns a nonzero value if there are events pending from the X server or alternate 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*;
    XtPointer *condition*;
    XtInputCallbackProc *proc*;
    XtPointer *client_data*;

source Specifies the source file descriptor on a POSIX-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 called when input is available.

client_data Specifies the parameter to be passed to *proc* when input is available.

The **XtAddInput** function registers in the 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*;
    XtPointer *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*, *client_data*)
    XtWorkProc *proc*;
    XtPointer *client_data*;

proc Procedure to call to do the work.

client_data Client data to pass to *proc* when it is called.

This routine registers a work procedure 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 any other resource specifications.

num_args Specifies the number of entries in *args*.

The procedure **XtCreateApplicationShell** calls **XtAppCreateShell** with *application_name* NULL, the application class passed to **XtInitialize**, and the default application context created by **XtInitialize**. This routine has been replaced by **XtAppCreateShell**.

An old-format resource type 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.

num_args Specifies the number of entries in *args*.

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 the size and pointer to the data in the *to* argument; 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 returned in *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.

The procedure type **XtConverter** has been replaced by **XtTypeConverter**.

The **XtStringConversionWarning** function is a convenience routine for old-format 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*’’. This routine has been superseded by **XtDisplayStringConversionWarning**.

To register an old-format converter, use **XtAddConverter** or **XtAppAddConverter**.

```
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 entries in *convert_args*.

XtAddConverter is equivalent in function to **XtSetTypeConverter** with *cache_type* equal to **XtCacheAll** for old-format type converters. It has been superseded by **XtSetTypeConverter.**

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 entries in *convert_args*.

XtAppAddConverter is equivalent in function to **XtAppSetTypeConverter** with *cache_type* equal to **XtCacheAll** for old-format type converters. It has been superseded by **XtAppSet-TypeConverter**.

To invoke resource conversions, a client may use **XtConvert** or, for old-format converters only, **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. oI

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 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 entries in *args*.
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**or **XtCallConverter**. 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.addr* for non-NULL. The data returned by **XtConvert** must be copied immediately by the caller, as it may point to static data in the type converter.

XtConvert has been replaced by **XtConvertAndStore**, and **XtDirectConvert** has been superseded by **XtCallConverter**.

To deallocate a shared GC when it is no longer needed, use **XtDestroyGC**.

void XtDestroyGC(*w*, *gc*)
  Widget *w*;
  GC *gc*;

w	Specifies any object on the display for which the shared GC was created. Must be of class Object or any subclass thereof.
gc	Specifies the shared 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 in the default application context 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_actions	Specifies the number of entries in *actions*.

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 **XtMenuPopup** and **XtMenuPopdown** as part of X Toolkit initialization. This routine has been replaced by **XtAppAddActions**. **XtInitialize** must be called before using this routine.

To set the Intrinsics selection timeout in the default application context, use **XtSetSelectionTimeout**.

void XtSetSelectionTimeout(*timeout*)
  unsigned long *timeout*;

timeout Specifies the selection timeout in milliseconds. This routine has been replaced by **XtAppSetSelectionTimeout**. **XtInitialize** must be called before using this routine.

To get the current selection timeout value in the default application context, use **XtGetSelection-Timeout**.

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 abort the selection request.

This routine has been replaced by **XtAppGetSelectionTimeout**. **XtInitialize** must be called before using this routine.

To obtain the global 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 **XtGetErrorDatbaseText**. This routine has been replaced by **XtAppGetErrorDatabase**.

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*)
  String *name*, *type*, *class*;
  String *default*;
  String *buffer_return*;
  int *nbytes*;

name
type Specify 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 associated with the default application context or returns the specified default message if one is not found in the error database. To form the full resource name and class when querying the database, the *name* and *type* are concatenated with a single ‘‘.’’ between them and the *class* is concatenated with itself with a single ‘‘.’’ if it does not already contain a ‘‘.’’. This routine has been superseded by **XtAppGetErrorDatabaseText**.

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 Intrinsics behavior is undefined. This routine has been superseded by **XtAppSetErrorMsgHandler**.

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 entries in *params*.

This routine has been superseded by **XtAppErrorMsg**.

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**. This routine has been superseded by **XtAppSetWarningMsgHandler**.

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 entries in *params*.

This routine has been superseded by **XtAppWarningMsg**.

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 POSIX-based systems, it prints the message to standard error and terminates the application. Fatal error message handlers should not return. If one does, subsequent X Toolkit behavior is undefined. This routine has been superseded by **XtAppSetErrorHandler**.

To call the installed fatal error procedure, use **XtError**.

void XtError(*message*)
    String *message*;

message Specifies the message to be reported.

Most programs should use **XtAppErrorMsg**, not **XtError**, to provide for customization and internationalization of error messages. This routine has been superseded by **XtAppError**.

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 POSIX-based systems, it prints the message to standard error and returns to the caller. This routine has been superseded by **XtAppSetWarningHandler**.

To call the installed nonfatal error procedure, use **XtWarning**.

void XtWarning(*message*)
    String *message*;

message Specifies the nonfatal error message to be reported.

Most programs should use **XtAppWarningMsg**, not **XtWarning**, to provide for customization and internationalization of warning messages. This routine has been superseded by **XtAppWarning**.

Appendix D

Intrinsics Error Messages

All Intrinsics errors and warnings have class ‘‘XtToolkitError’’. The following two tables summarize the common errors and warnings that can be generated by the Intrinsics. Additional implementation-dependent messages are permitted.

Error Messages

Name	Type	Default Message
allocError	calloc	Cannot perform calloc
allocError	malloc	Cannot perform malloc
allocError	realloc	Cannot perform realloc
communicationError	select	Select failed
internalError	shell	Shell’s window manager interaction is broken
invalidArgCount	xtGetValues	Argument count > 0 on NULL argument list in XtGetValues
invalidArgCount	xtSetValues	Argument count > 0 on NULL argument list in XtSetValues
invalidClass	constraintSetValue	Subclass of Constraint required in CallConstraintSetValues
invalidClass	xtAppCreateShell	XtAppCreateShell requires non-NULL widget class
invalidClass	xtCreatePopupShell	XtCreatePopupShell requires non-NULL widget class
invalidClass	xtCreateWidget	XtCreateWidget requires non-NULL widget class
invalidClass	xtPopdown	XtPopdown requires a subclass of shellWidgetClass
invalidClass	xtPopup	XtPopup requires a subclass of shellWidgetClass
invalidDimension	xtCreateWindow	Widget %s has zero width and/or height
invalidDimension	shellRealize	Shell widget %s has zero width and/or height
invalidDisplay	xtInitialize	Can’t Open display
invalidGeometryManager	xtMakeGeometryRequest	XtMakeGeometryRequest - parent has no geometry manger
invalidParameter	removePopupFromParent	RemovePopupFromParent requires non-NULL popuplist
invalidParameter	xtAddInput	invalid condition passed to XtAddInput
invalidParameters	xtMenuPopupAction	MenuPopup wants exactly one argument
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
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
translationError	mergingTablesWithCycles	Trying to merge translation tables with cycles, and can’t resolve this cycle.

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	xtAddGrab	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
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
invalidParameters	xtMenuPopdown	XtMenuPopdown called with num_params != 0 or 1
invalidParent	xtCopyFromParent	CopyFromParent must have non-NULL parent
invalidPopup	xtMenuPopup	Can’t find popup in _XtMenuPopup
invalidPopup	xtMenuPopdown	Can’t find popup in _XtMenuPopdown
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
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
missingCharsetList	cvtStringToFontSet	Missing charsets in String to FontSet conversion
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-existant accelerators
translationError	ambiguousActions	Overriding earlier translation manager actions.
translationError	mergingNullTable	Old translation table was null, cannot modify.
translationError	nullTable	Can't translate event thorugh NULL table
translationError	unboundActions	Actions not found: %s
translationError	xtTranslateInitialize	Initializing Translation manager twice.
translationParseError	showLine	... found while parsing '%s'
translationParseError	parseError	translation table syntax error: %s
translationParseError	parseString	Missing '´.
typeConversionError	noConverter	No type converter registered for '%s' to '%s' conversion.
versionMismatch	widget	Widget class %s version mismatch:0idget %d vs. intrinsics %d.
wrongParameters	cvtIntOrPixelToXColor	Pixel to color conversion needs screen and colormap arguments
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
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	cvtStringToCursor	String to cursor conversion needs screen argument
wrongParameters	cvtStringToDisplay	String to Display conversion needs no extra arguments
wrongParameters	cvtStringToFile	String to File conversion needs no extra arguments
wrongParameters	cvtStringToFont	String to font conversion needs screen argument
wrongParameters	cvtStringToFontSet	String to FontSet conversion needs display and locale arguments
wrongParameters	cvtStringToFontStruct	String to cursor conversion needs screen argument
wrongParameters	cvtStringToInt	String to Integer conversion needs no extra arguments
wrongParameters	cvtStringToPixel	String to pixel conversion needs screen and colormap 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

Appendix E

Defined Strings

The **StringDefs.h** header file contains definitions for the following resource name, class, and representation type symbolic constants.

Resource names:

Symbol	Definition
XtNaccelerators	"accelerators"
XtNallowHoriz	"allowHoriz"
XtNallowVert	"allowVert"
XtNancestorSensitive	"ancestorSensitive"
XtNbackground	"background"
XtNbackgroundPixmap	"backgroundPixmap"
XtNbitmap	"bitmap"
XtNborderColor	"borderColor"
XtNborder	"borderColor"
XtNborderPixmap	"borderPixmap"
XtNborderWidth	"borderWidth"
XtNcallback	"callback"
XtNchildren	"children"
XtNcolormap	"colormap"
XtNdepth	"depth"
XtNdestroyCallback	"destroyCallback"
XtNeditType	"editType"
XtNfile	"file"
XtNfont	"font"
XtNfontSet	"fontSet"
XtNforceBars	"forceBars"
XtNforeground	"foreground"
XtNfunction	"function"
XtNheight	"height"
XtNhighlight	"highlight"
XtNhSpace	"hSpace"
XtNindex	"index"
XtNinitialResourcesPersistent	"initialResourcesPersistent"
XtNinnerHeight	"innerHeight"
XtNinnerWidth	"innerWidth"
XtNinnerWindow	"innerWindow"
XtNinsertPosition	"insertPosition"
XtNinternalHeight	"internalHeight"
XtNinternalWidth	"internalWidth"
XtNjumpProc	"jumpProc"
XtNjustify	"justify"
XtNknobHeight	"knobHeight"
XtNknobIndent	"knobIndent"
XtNknobPixel	"knobPixel"
XtNknobWidth	"knobWidth"

XtNlabel	"label"
XtNlength	"length"
XtNlowerRight	"lowerRight"
XtNmappedWhenManaged	"mappedWhenManaged"
XtNmenuEntry	"menuEntry"
XtNname	"name"
XtNnotify	"notify"
XtNnumChildren	"numChildren"
XtNorientation	"orientation"
XtNparameter	"parameter"
XtNpixmap	"pixmap"
XtNpopupCallback	"popupCallback"
XtNpopdownCallback	"popdownCallback"
XtNresize	"resize"
XtNreverseVideo	"reverseVideo"
XtNscreen	"screen"
XtNscrollProc	"scrollProc"
XtNscrollDCursor	"scrollDCursor"
XtNscrollHCursor	"scrollHCursor"
XtNscrollLCursor	"scrollLCursor"
XtNscrollRCursor	"scrollRCursor"
XtNscrollUCursor	"scrollUCursor"
XtNscrollVCursor	"scrollVCursor"
XtNselection	"selection"
XtNselectionArray	"selectionArray"
XtNsensitive	"sensitive"
XtNshown	"shown"
XtNspace	"space"
XtNstring	"string"
XtNtextOptions	"textOptions"
XtNtextSink	"textSink"
XtNtextSource	"textSource"
XtNthickness	"thickness"
XtNthumb	"thumb"
XtNthumbProc	"thumbProc"
XtNtop	"top"
XtNtranslations	"translations"
XtNunrealizeCallback	"unrealizeCallback"
XtNupdate	"update"
XtNuseBottom	"useBottom"
XtNuseRight	"useRight"
XtNvalue	"value"
XtNvSpace	"vSpace"
XtNwidth	"width"
XtNwindow	"window"
XtNx	"x"
XtNy	"y"

Resource classes:

Symbol	Definition
XtCAccelerators	"Accelerators"
XtCBackground	"Background"

XtCBitmap	"Bitmap"
XtCBoolean	"Boolean"
XtCBorderColor	"BorderColor"
XtCBorderWidth	"BorderWidth"
XtCCallback	"Callback"
XtCColormap	"Colormap"
XtCColor	"Color"
XtCCursor	"Cursor"
XtCDepth	"Depth"
XtCEditType	"EditType"
XtCEventBindings	"EventBindings"
XtCFile	"File"
XtCFont	"Font"
XtCFontSet	"FontSet"
XtCForeground	"Foreground"
XtCFraction	"Fraction"
XtCFunction	"Function"
XtCHeight	"Height"
XtCHSpace	"HSpace"
XtCIndex	"Index"
XtCInitialResourcesPersistent	"InitialResourcesPersistent"
XtCInsertPosition	"InsertPosition"
XtCInterval	"Interval"
XtCJustify	"Justify"
XtCKnobIndent	"KnobIndent"
XtCKnobPixel	"KnobPixel"
XtCLabel	"Label"
XtCLength	"Length"
XtCMappedWhenManaged	"MappedWhenManaged"
XtCMargin	"Margin"
XtCMenuEntry	"MenuEntry"
XtCNotify	"Notify"
XtCOrientation	"Orientation"
XtCParameter	"Parameter"
XtCPixmap	"Pixmap"
XtCPosition	"Position"
XtCReadOnly	"ReadOnly"
XtCResize	"Resize"
XtCReverseVideo	"ReverseVideo"
XtCScreen	"Screen"
XtCScrollProc	"ScrollProc"
XtCScrollDCursor	"ScrollDCursor"
XtCScrollHCursor	"ScrollHCursor"
XtCScrollLCursor	"ScrollLCursor"
XtCScrollRCursor	"ScrollRCursor"
XtCScrollUCursor	"ScrollUCursor"
XtCScrollVCursor	"ScrollVCursor"
XtCSelection	"Selection"
XtCSensitive	"Sensitive"
XtCSelectionArray	"SelectionArray"
XtCSpace	"Space"
XtCString	"String"
XtCTextOptions	"TextOptions"
XtCTextPosition	"TextPosition"

XtCTextSink	"TextSink"
XtCTextSource	"TextSource"
XtCThickness	"Thickness"
XtCThumb	"Thumb"
XtCTranslations	"Translations"
XtCValue	"Value"
XtCVSpace	"VSpace"
XtCWidth	"Width"
XtCWindow	"Window"
XtCX	"X"
XtCY	"Y"

Resource representation types:

Symbol	Definition
XtRAcceleratorTable	"AcceleratorTable"
XtRAtom	"Atom"
XtRBitmap	"Bitmap"
XtRBool	"Bool"
XtRBoolean	"Boolean"
XtRCallback	"Callback"
XtRCallProc	"CallProc"
XtRCardinal	"Cardinal"
XtRColor	"Color"
XtRColormap	"Colormap"
XtRCursor	"Cursor"
XtRDimension	"Dimension"
XtRDisplay	"Display"
XtREditMode	"EditMode"
XtREnum	"Enum"
XtRFile	"File"
XtRFloat	"Float"
XtRFont	"Font"
XtRFontSet	"FontSet"
XtRFontStruct	"FontStruct"
XtRFunction	"Function"
XtRGeometry	"Geometry"
XtRImmediate	"Immediate"
XtRInitialState	"InitialState"
XtRInt	"Int"
XtRJustify	"Justify"
XtRLongBoolean	XtRBool
XtRObject	"Object"
XtROrientation	"Orientation"
XtRPixel	"Pixel"
XtRPixmap	"Pixmap"
XtRPointer	"Pointer"
XtRPosition	"Position"
XtRScreen	"Screen"
XtRShort	"Short"
XtRString	"String"
XtRStringArray	"StringArray"
XtRStringTable	"StringTable"

XtRUnsignedChar	"UnsignedChar"
XtRTranslationTable	"TranslationTable"
XtRVisual	"Visual"
XtRWidget	"Widget"
XtRWidgetClass	"WidgetClass"
XtRWidgetList	"WidgetList"
XtRWindow	"Window"

Boolean enumeration constants:

Symbol	Definition
XtEoff	"off"
XtEfalse	"false"
XtEno	"no"
XtEon	"on"
XtEtrue	"true"
XtEyes	"yes"

Orientation enumeration constants:

Symbol	Definition
XtEvertical	"vertical"
XtEhorizontal	"horizontal"

Text edit enumeration constants:

Symbol	Definition
XtEtextRead	"read"
XtEtextAppend	"append"
XtEtextEdit	"edit"

Color enumeration constants:

Symbol	Definition
XtExtdefaultbackground	"xtdefaultbackground"
XtExtdefaultforeground	"xtdefaultforeground"

Font constant:

Symbol	Definition
XtExtdefaultfont	"xtdefaultfont"

Table of Contents