Programming With the Xt Intrinsics

Version 11

HP 9000 Series 300/800 Computers

HP Part Number 98794-90000

HEWLETT PACKARD

Hewlett-Packard Company
1000 NE Circle Boulevard, Corvallis, Oregon 97330-9988

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1988...Release 1.

December 1988...Release 2.

May 1989...Update. This Update replaces the "Programming With the Xt Intrinsics" section.

June 1989...Release 3. This edition incorporates the May 1989 Update.

This page left blank intentionally.

Contents

X Toolkit Overview 1

The X Toolkit provides the base functionality necessary to build a wide variety of application environments. It is fully extensible and supportive of the independent development of new or extended components. This is accomplished by defining interfaces that mask implementation details from both applications and common component implementors. By following a small set of conventions, a programmer can extend the X Toolkit in new ways and have these extensions function smoothly with the existing facilities.

The X Toolkit is a library package layered on top of the X Window System. This layer extends the basic abstractions provided by X and, thus, provides the next layer of functionality by supplying mechanisms for intercomponent and intracomponent interactions. In the X Toolkit, a widget is a combination of an X window (or subwindow) and its associated semantics.

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

- Policy
- Consistency
- Style

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

1.1 Introduction

The X Toolkit provides tools that simplify the design of application user interfaces in the X Window System programming environment. It assists application programmers by providing a commonly used set of underlying user-interface functions to manage:

- Toolkit initialization
- Widgets
- Memory
- Window, file, and timer events

- Widget geometry
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmaps
- Errors and warnings

At present, the X Toolkit consists of:

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

The Intrinsics mechanisms are intended for the widget programmer. The architectural model lets the widget programmer design new widgets by using the Intrinsics or by combining other widgets. The application interface layers built on top of the X Toolkit include a coordinated set of widgets and composition policies. Some of these widgets and policies are application domain specific, while others are common across a number of application domains.

The X Toolkit provides an architectural model that is flexible enough to accommodate a number of different application interface layers. In addition, the supplied set of X Toolkit functions are:

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

Applications that use the X Toolkit must include the following header files:

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

and possibly also:

- `<X11/Xatoms.h>`

- `<X11/Shell.h>`

Widget implementations should include

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

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

1.2 Terminology

The following terms are used throughout this manual.

`Application programmer`
: A programmer who uses the X Toolkit to produce an application user interface.

`Class`
: The general group that a specific object belongs to.

`Client`
: A routine that uses a widget in an application or for composing another widget.

`Instance`
: A specific widget object as opposed to a general widget class.

`Method`
: The functions or procedures that a widget itself implements.

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

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

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

`User`

A person interacting with a workstation.

`Widget`

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

`Widget class`

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

`Widget programmer`

A programmer who adds new widgets to the X Toolkit.

Widgets 2

The fundamental data type of the X Toolkit is the widget, which is dynamically allocated and contains state information. Every widget belongs to exactly one "widget class" that is statically allocated and initialized and that contains the operations allowable on widgets of that class.

Logically, a widget is a rectangle with associated input/output semantics. Some widgets display information (for example, text or graphics), while 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, while others change their display in response to input and can invoke functions that an application has attached to them.

Much of the input/output of a widget is customizable by users. Such customization includes fonts, colors, sizes, border widths, and so on.

A widget instance is composed of two parts:

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

Logically, a widget class is the procedures and data that is associated with all widgets belonging to that class. These procedures and data can be inherited by subclasses.

Physically, a widget class is a pointer to a structure. The contents of this structure are constant for all widgets of the widget class, even though the values can vary from widget class to widget class. (Here, "constant" means the class structure is initialized at compile-time and never changed, except for a one-shot class initialization and in-place compilation of resource lists, which takes place when the first widget of the class or subclass is created.) A widget instance is allocated and initialized by `XtCreateWidget`. For further information, see "Creating Widgets".

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

2.1 Core Widget Definition

The Core widget contains the definitions of fields common to all widgets. All widgets are subclasses of Core.

2.1.1 CoreClassPart

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

```
typedef struct {
      WidgetClass superclass;       See ''Widget Classes''
      String class_name;            See ''Widget Classes''
      Cardinal widget_size;         See ''Creating Widgets''
      XtProc class_initialize;      See ''Widget Classes''
      XtWidgetClassProc class_part_initialize;See ''Widget Classes''
      Boolean class_inited;         Private to ''XtCreateWidget''
      XtInitProc initialize;        See ''Creating Widgets''
      XtArgsProc initialize_hook;   See ''Creating Widgets''
      XtRealizeProc realize;        See ''Creating Widgets''
      XtActionList actions;         See ''Translation Management''
      Cardinal num_actions;         See ''Translation Management''
      XtResourceList resources;     See ''Resource Management''
      Cardinal num_resources;       See ''Resource Management''
      XrmClass xrm_class;           Private to ''Resource Management''
      Boolean compress_motion;      See ''Mouse Motion Compression''
      Boolean compress_exposure;    See ''Exposure Compression''
      Boolean compress_enterleave;  See ''Enter/Leave Compression''
      Boolean visible_interest;     See ''Widget Exposure and Visibility''
      XtWidgetProc destroy;         See ''Destroying Widgets''
      XtWidgetProc resize;          See ''Geometry Management''
      XtExposeProc expose;          See ''Widget Exposure and Visibility''
      XtSetValuesFunc set_values;   See ''Reading and Writing Widget State''
      XtArgsFunc set_values_hook;   See ''Reading and Writing Widget State''
      XtAlmostProc set_values_almost;See ''Reading and Writing Widget State''
      XtArgsProc get_values_hook;   See ''Reading and Writing Widget State''
      XtWidgetProc accept_focus;    See ''Focus Management''
      XtVersionType version;        See ''Widget Classes''
      _XtOffsetList callback_private;Private to ''Callbacks''
      String tm_table;              See ''Translation Management''
      XtGeometryHandler query_geometry;See ''Geometry Management''
} CoreClassPart;
```

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

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

The predefined class record and pointer for WidgetClassRec are:

```
extern WidgetClassRec widgetClassRec;

extern WidgetClass widgetClass;
```

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

2.1.2 CorePart

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

```
typedef struct {
      Widget self;
      WidgetClass widget_class;     See ''Widget Classes''
      Widget parent;                See ''Widget Classes''
      String name;                  See ''Resource Management''
      XrmName xrm_name;             Private to ''Resource Management''
      Screen *screen;               See ''Obtaining Window Information''
      Colormap colormap;            See ''Obtaining Window Information''
      Window window;                See ''Obtaining Window Information''
      Position x;                   See ''Geometry Management''
      Position y;                   See ''Geometry Management''
      Dimension width;              See ''Geometry Management''
      Dimension height;             See ''Geometry Management''
      Cardinal depth;               See ''Window Attributes''
      Dimension border_width;       See ''Geometry Management''
      Pixel border_pixel;           See ''Obtaining Window Information''
      Pixmap border_pixmap;         See ''Obtaining Window Information''
      Pixel background_pixel;       See ''Obtaining Window Information''
      Pixmap background_pixmap;     See ''Obtaining Window Information''
      _XtEventTable event_table;    Private to ''Event Management''
      struct _TMRec tm;             Private to ''Translation Management''
      caddr_t constraints;          See ''Constrained Composite Widgets''
      Boolean visible;              See ''Widget Visibility and Exposure''
      Boolean sensitive;            See ''Setting and Checking Sensitivity''
      Boolean ancestor_sensitive;   See ''Setting and Checking Sensitivity''
      Boolean managed;              See ''Composite Widgets''
      Boolean mapped_when_managed;  See ''Composite Widgets''
      Boolean being_destroyed;      See ''Destroying Widgets''
      XtCallbackList destroy_callbacks;See ''Destroying Widgets''
      WidgetList popup_list;        See''Pop-up Widgets''
      Cardinal num_popups;          See ''Pop-up Widgets''
} CorePart;
```

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

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

2.1.3 CorePart Default Values

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

Field	Default Value
self	address of the widget structure (may not be changed)
widget_class	widget_class argument to `XtCreateWidget` (may not be changed)
parent	parent argument to `XtCreateWidget` (may not be changed)
name	name argument to `XtCreateWidget` (may not be changed)
screen	parent's screen, but top-level widget from display specifier (may not be changed)
colormap	the default color map for the screen
window	NULL
x	0
y	0
width	0
height	0
depth	parent's depth, but top-level widget gets root window depth
border_width	1
border_pixel	BlackPixel of screen
border_pixmap	NULL
background_pixel	WhitePixel of screen
background_pixmap	NULL
visible	TRUE
sensitive	TRUE
ancestor_sensitive	bitwise AND of parent's sensitive & ancestor_sensitive
managed	FALSE
map_when_managed	TRUE
being_destroyed	parent's being_destroyed
destroy_callbacks	NULL

2.2 Composite Widget Definition

Composite widgets are a subclass of the Core widget and are more fully described in "Composite Widgets".

2.2.1 CompositeClassPart

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

```
typedef struct {
      XtGeometryHandler geometry_manager;See ''Geometry Management''
      XtWidgetProc change_managed;   See ''Composite Widgets''
      XtWidgetProc insert_child;     See ''Composite Widgets''
      XtWidgetProc delete_child;     See ''Composite Widgets''
      XtWidgetProc move_focus_to_next;See ''Focus Management''
      XtWidgetProc move_focus_to_prev;See ''Focus Management''
} CompositeClassPart;
```

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

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

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

```
extern CompositeClassRec compositeClassRec;

extern WidgetClass compositeWidgetClass;
```

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

2.2.2 CompositePart

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

```
typedef struct {
      WidgetList children;          See ''Widget Classes''
      Cardinal num_children;        See ''Widget Classes''
      Cardinal num_slots;           See ''Composite Widgets''
      Cardinal num_mapped_children; See ''Composite Widgets''
      XtOrderProc insert_position;  See ''Creating Widgets''
} CompositePart;
```

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

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

2.2.3 CompositePart Default Values

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

Field	Default Value
children	NULL
num_children	0
num_slots	0
num_mapped_children	0
insert_position	internal function InsertAtEnd

2.3 Constraint Widget Definition

Constraint widgets are a subclass of the Composite widget and are more fully described in “Constrained Composite Widgets”.

2.3.1 ConstraintClassPart

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

```
typedef struct {
      XtResourceList resources;      See ''Constrained Composite Widgets''
      Cardinal num_resources;        See ''Constrained Composite Widgets''
      Cardinal constraint_size;      See ''Constrained Composite Widgets''
      XtInitProc initialize;         See ''Constrained Composite Widgets''
      XtWidgetProc destroy;          See ''Constrained Composite Widgets''
      XtSetValuesFunc set_values;    See ''Constrained Composite Widgets''
} ConstraintClassPart;
```

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

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

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

```
extern ConstraintClassRec constraintClassRec;

extern WidgetClass constraintWidgetClass;
```

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

2.3.2 ConstraintPart

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

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

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

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

This page left blank intentionally.

Widget Classes 3

The widget_class field of a widget points to its widget class structure. This structure contains information that is constant across all widgets of that class.

This class-oriented structure means that widget classes do not usually implement directly callable procedures. Rather, they implement procedures that are available through their widget class structure. These class procedures are invoked by generic procedures that envelop common actions around the procedures implemented by the widget class. Such procedures are applicable to all widgets of that class and also to widgets that are subclasses of that class.

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

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

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

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

3.1 Widget Naming Conventions

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

- Use the X naming conventions that are applicable. For example, a record component name is all lower-case and uses underscore (_) for compound words (for example, background_pixmap). Type and procedure names start with upper-case and use capitalization for compound words (for example, XtArgList or XtSetValues).

- A resource name string is spelled identically to the field name, except that compound names use capitalization rather than underscore. To let the compiler catch spelling errors, each resource name should have a macro definition prefixed with XtN. For example, the background_pixmap field has the corresponding resource name identifier XtNbackgroundPixmap, which is defined as the string "backgroundPixmap". Many predefined names are listed in the <X11/StringDefs.h> header file. Before you invent a new name, you should make sure that your proposed name is not already defined or that there already is not name that you can use.

- A resource class string starts with a capital letter, and uses capitalization for compound names (for example,"BorderWidth"). Each resource class string should have a macro definition prefixed with XtC (for example, XtCBorderWidth).

- A resource representation string is spelled identically to the type name (for example, "TranslationTable"). Each representation string should have a macro definition prefixed with XtR (for example, XtRTranslationTable).

- New widget classes start with a capital and use capitalization for compound words. Given a new class name "AbcXyz" you should derive several names:

  - Partial widget instance structure name AbcXyzPart
  - Complete widget instance structure names AbcXyzRec and _AbcXyzRec
  - Widget instance pointer type name AbcXyzWidget
  - Partial class structure name AbcXyzClassPart
  - Complete class structure names AbcXyzClassRec and _AbcXyzClassRec
  - Class structure variable abcXyzClassRec
  - Class pointer variable abcXyzWidgetClass

- Action procedures available to translation specifications should follow the same naming conventions as procedures. That is, they start with a capital letter and compound names use capitalization. For example, "Highlight" and "NotifyClient".

3.2 Widget Subclassing in Public ".h" Files

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

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

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

```
/**************************************************************************
 *
 * StaticText Widget
 *
 **************************************************************************/
extern WidgetClass XwstatictextWidgetClass;

typedef struct _XwStaticTextClassRec*XwStaticTextWidgetClass;
typedef struct _XwStaticTextRec     *XwStaticTextWidget;
```

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

3.3 Widget Subclassing in Private ".h" Files

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

- A reference to the public ".h" file for the class.
- A reference to the private ".h" file for the superclass.
- The new fields that the widget instance adds to its superclass's widget structure.
- The complete widget instance structure for this widget.
- The new fields that this widget class adds to its superclass's `Constraint` structure, if the widget is a subclass of `Constraint`.

- The complete `Constraint` structure, if the widget is a subclass of `Constraint`.
- The new fields that this widget class adds to its superclass's widget class structure.
- The complete widget class structure for this widget.
- The name of a "constant" of the generic widget class structure.
- For each new procedure in the widget class structure, an "InheritOperation" procedure for subclasses that wish to merely inherit a superclass operation.

For example, the following is the private ".h" file for the StaticText widget:

```
/********************************************
 *
 *   No new fields need to be defined
 *   for the StaticText widget class record
 *
 ********************************************/
typedef struct {int foo;} XwStaticTextClassPart;

/****************************************************
 *
 * Full class record declaration for StaticText class
 *
 ****************************************************/
typedef struct _XwStaticTextClassRec {
      CoreClassPart                  core_class;
      XwPrimitiveClassPart           primitive_class;
      XwStaticTextClassPart          statictext_class;
} XwStaticTextClassRec;

/********************************************
 *
 * New fields needed for instance record
 *
 ********************************************/
typedef struct _XwStaticTextPart {
      /*
       * "Public" members (Can be set by resource manager).
       */
      char *input_string;            /* String sent to this widget. */
      XwAlignment alignment;         /* Alignment within the box */
      int gravity;                   /* Controls use of extra space in window */
      Boolean wrap;                  /* Controls wrapping on spaces */
      Boolean strip;                 /* Controls stripping of blanks */
      int line_space;                /* Ratio of font height use as dead space
                                        between lines.  Can be less than zero
                                        but not less than -1.0  */
      XFontStruct *font;             /* Font to write in. */
      Dimension internal_height;     /* Space from text to top and
                                          bottom highlights */
      Dimension internal_width;      /* Space from left and right side
                                          highlights */
      /*
       * "Private" members -- values computed by
       *  XwStaticTextWidgetClass methods.
       */
      GC normal_GC;                /* GC for text */
      XRectangle TextRect;         /* The bounding box of the text,
                                      or clip rectangle of the window;
                                      whichever is smaller. */
      char *output_string;         /* input_string after formatting */
} XwStaticTextPart;

/*****************************************************************
 *
 * Full instance record declaration
 *
 *****************************************************************/
typedef struct _XwStaticTextRec {
      CorePart          core;
      XwPrimitivePart   primitive;
      XwStaticTextPart  static_text;
} XwStaticTextRec;
```

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

3.4 Widget Subclassing in ".c" Files

The ".c" file for a widget contains the structure initializer for the class record variable. This initializer can be broken up into several parts:

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

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

The class_name field contains the text name for this class (used by the resource manager). For example, the StaticText widget has the string "StaticText".

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

The version field indicates the toolkit version number and is used for runtime consistency checking of the X Toolkit and widgets in an application. Widget writers must set it to the symbolic value `XtVersion` in the widget class initialization.

All other fields are described in their respective sections.

The following is a somewhat compressed version of the ".c" file for the StaticTextwidget. (The "resources" table is described in the section "Resource Management").

```
/*************************************<->*************************************
 *
 *
 *    Description:  resource list for class: StaticText
 *    -----------
 *
 *    Provides default resource settings for instances of this class.
 *    To get full set of default settings, examine resouce list of super
 *    classes of this class.
 *
 *************************************<->***********************************/
```

```
static XtResource resources[] = {
      { XtNhSpace,
            XtCHSpace,
            XtRInt,
            sizeof(int),
            XtOffset(XwStaticTextWidget, static_text.internal_width),
            XtRString,
            "2"
      },
      { XtNvSpace,
            XtCVSpace,
            XtRInt,
            sizeof(int),
            XtOffset(XwStaticTextWidget, static_text.internal_height),
            XtRString,
            "2"
      },
      { XtNalignment,
            XtCAlignment,
            XtRAlignment,
            sizeof(XwAlignment),
            XtOffset(XwStaticTextWidget,static_text.alignment),
            XtRString,
            "Left"
      },
      { XtNgravity,
            XtCGravity,
            XtRGravity,
            sizeof(int),
            XtOffset(XwStaticTextWidget,static_text.gravity),
            XtRString,
            "CenterGravity"
      },
      .
      .
      .

/*************************************<->*************************************
 *
 *
 *    Description:  global class record for instances of class: StaticText
 *    -----------
 *
 *    Defines default field settings for this class record.
 *
 *************************************<->***********************************/

XwStaticTextClassRec XwstatictextClassRec = {
      { /* core_class fields */
      /* superclass            */    (WidgetClass) &XwprimitiveClassRec,
      /* class_name            */    "StaticText",
      /* widget_size           */    sizeof(XwStaticTextRec),
      /* class_initialize      */    ClassInitialize,
      /* class_part_initialize */    NULL,
      /* class_inited          */    FALSE,
      /* initialize            */    (XtInitProc) Initialize,
      /* initialize_hook       */    NULL,
    /* realize            */    (XtRealizeProc) Realize,
    /* actions            */    actionsList,
    /* num_actions        */    XtNumber(actionsList),
    /* resources          */    resources,
    /* num_resources      */    XtNumber(resources),
    /* xrm_class          */    NULLQUARK,
    /* compress_motion    */    TRUE,
    /* compress_exposure  */    TRUE,
    /* compress_enterleave */   TRUE,
    /* visible_interest   */    FALSE,
    /* destroy            */    (XtWidgetProc) Destroy,
    /* resize             */    (XtWidgetProc) Resize,
    /* expose             */    (XtExposeProc) Redisplay,
    /* set_values         */    (XtSetValuesFunc) SetValues,
    /* set_values_hook    */    NULL,
    /* set_values_almost  */    (XtAlmostProc) XtInheritSetValuesAlmost,
    /* get_values_hook    */    NULL,
    /* accept_focus       */    NULL,
    /* version            */    XtVersion,
    /* callback private   */    NULL,
    /* tm_table           */    defaultTranslations,
    /* query_geometry     */    NULL,
    },
    {
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
    }
};

/* Class record pointer */
WidgetClass XwstatictextWidgetClass = (WidgetClass) &XwstatictextClassRec;
```

3.5 Initialization of a Class

Many class records can be initialized completely at compile time. But in some cases, a class may want to register type converters or perform other sorts of “one-shot” 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 X Toolkit. A class initialization procedure is of type `XtProc`:

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

void Proc()
```

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

In addition to doing class initializations that get done exactly once, some classes need to perform additional initialization for fields in its part of the class record. These get done not just for the particular class but for subclasses as well. This is done in the class's class part initialization procedure. The class part initialization procedure is of type `XtClassProc:`

```
typedef void (*XtClassProc)();

void ClassProc(widgetClass)
      WidgetClass widgetClass;
```

During class initialization, the class part initialization procedure for the class and all its superclasses are called in a superclass to subclass order on the class record. These procedures have the responsibility of doing any dynamic initializations necessary to their class's part of the record. The most common is the resolution of any inherited methods defined in the class. For example, if a widget class C has superclasses Core, Composite, A, and B, the class record for C first is passed to Core's class_part_initialize record. This resolves any inherited core methods and compiles the textual representations of the resource list and action table that are defined in the class record. Next, the Composite's class_part_initialize is called to initialize the composite part of C's class record. Finally, the class_part_initialize procedures for A, B, and C (in order) are called. For further information, see "Inheriting Superclass Operations". 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) should start off with their class_inited field FALSE.

The first time a widget of that 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, calling the class_initialize and the class_part_initialize procedures for the class and all its superclasses. The class_inited field is then set to TRUE. After the one-time initialization, a class structure is constant.

```
/*************************************<->*************************************
 *
 *  ClassInitialize
 *
 *   Description:
 *   -----------
 *    Set fields in primitive class part of our class record so that
 *    the traversal code can invoke our select/release procedures (note
 *    that for this class toggle is a empty proc).
 *
 *************************************<->***********************************/
static void ClassInitialize()
{
      XwstatictextClassRec.primitive_class.select_proc = (XtWidgetProc) Select;
      XwstatictextClassRec.primitive_class.release_proc = (XtWidgetProc) Release;
      XwstatictextClassRec.primitive_class.toggle_proc = (XtWidgetProc) Toggle;
}
```

3.6 Obtaining the Class and Superclass of a Widget

To obtain the class of a widget, use XtClass.

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

w   Specifies the widget.

XtClass 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.

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

3.7 Verifying the Subclass of a Widget

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

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

w Specifies the widget under question.

widget_class Specifies the widget class to test against.

`XtIsSubclass` returns TRUE if the class of the specified widget w is equal to or is a subclass of widget_class. The specified widget w may be arbitrarily far down the subclass chain; it need not be an immediate subclass of widget_class. Composite widgets that wish to restrict the class of the items they contain can use `XtIsSubclass` to find out if a widget belongs to the desired class of objects.

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

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

w Specifies the widget under question.

widget_class Specifies the widget class to test against.

message Specifies an error message.

`XtCheckSubclass` determines if the class of the specified w is equal to or is a subclass of widget_class. Again, w may be any number of subclasses down the chain and need not be an immediate subclass of widget_class. If w is not a subclass, `XtCheckSubclass` constructs an error message from the supplied message, the widget's actual class, and the expected class. Then, it calls `XtError`. `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 including and linking against the debugging version of the Intrinsics. Otherwise, it is defined as the empty string and so generates no code.

3.8 Superclass Chaining

Some fields defined in the widget class structure are self-contained and are independent of the values for these fields defined in superclasses. Among these are:

- class_name
- widget_size
- realize

- visible_interest
- resize
- expose
- accept_focus
- compress_motion
- compress_exposure
- compress_enterleave
- set_values_almost
- tm_table
- version

Some fields defined in the widget class structure make sense only after their superclass has been operated on. In this case, the invocation of a single operation actually first accesses the Core class, then the subclass, and so on down the class chain to the widget class of the widget. These superclass-to-subclass fields are:

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

For subclasses of `Constraint`, the constraint resources field is chained from the `Constraint` class down to the subclass.

Some fields defined in the widget class structure make sense only after their subclass has been operated on. In this case, the invocation of a single operation actually first accesses the widget class, then its superclass, and so on up the class chain to the Core class. The subclass-to-superclass fields are:

- destroy
- actions

3.9 Inheriting 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

To inherit an operation "xyz", you simply specify the procedure `XtInheritXyz` in your class record.

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

Inheriting works by comparing the value of the field with a known, special value and by copying in the superclass's value for that field if a match occurs. This special value is usually the intrinsic routine `_XtInherit` cast to the appropriate type.

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

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

The Composite's class_part_initialize procedure begins:

```
static void CompositeClassPartInitialize(widgetClass)
      WidgetClass widgetClass;
{
      register CompositeWidgetClass wc = (CompositeWidgetClass) widgetClass;
      CompositeWidgetClass super = (CompositeWidgetClass) wc->core.class.superclass

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

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

The inherit procedures defined for Core are:

- XtInheritRealize
- XtInheritResize
- XtInheritExpose
- XtInheritSetValuesAlmost
- XtInheritAcceptFocus

The inherit procedures defined for Composite are:

- XtInheritGeometryManager
- XtInheritChangeManaged
- XtInheritInsertChild
- XtInheritDeleteChild
- XtInheritMoveFocusToNext
- XtInheritMoveFocusToPrev

3.10 Calling Superclass Operations

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

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

The following is abbreviated code for a possible implementation of a Shell's insert_child procedure:

```
static void InsertChild(w)
      Widget w;
{
      (*(((CompositeWidgetClass)XtSuperclass(shellWidgetClass))
            ->composite_class.insert_child)) (w);
      XtManageChild(w);                /* Add to managed set now */
}
```

This page left blank intentionally.

Instantiating Widgets 4

Widgets are either "primitive" or "composite". Either kind of widget can have "pop-up" children widgets, but only composite widgets can have "normal" children widgets. A composite widget may in unusual circumstances have zero normal children but usually has at least one. Widgets with no children of any kind are leaves of a widget tree. Widgets with one or more children are intermediate nodes of a tree. The shell widget returned by `XtInitialize` or `XtCreateApplicationShell` is the root of a widget tree.

The "normal" children of the widget tree exactly duplicates the associated window tree. Each pop-up child has a window which is a child of the root window so that the pop-up window is not clipped. Again, the normal children of a pop-up exactly duplicates the window tree associated with the pop-up window.

A widget tree is manipulated by several X Toolkit functions. For example, `XtRealizeWidget` traverses the tree downward to recursively realize normal children widgets. `XtDestroyWidget` traverses the tree downward to destroy all children. The functions that fetch and modify resources traverse the tree upward to determine the inheritance of resources from a widget's ancestors. `XtMakeGeometryRequest` traverses the tree one level upward to get the geometry manager responsible for a normal widget child's geometry.

To facilitate up-traversal of the widget tree, each widget has a pointer to its parent widget. Shell widgets returned by `XtInitialize` and `XtCreateApplicationShell` have a parent pointer of NULL.

To facilitate down-traversal of the widget tree, each composite widget has a pointer to an array of children widgets. This array includes all normal children created, not just the subset of children that are managed by the composite widget's geometry manager.

In addition, every widget has a pointer to an array of pop-up children widgets.

4.1 Initializing the X Toolkit

Before any of the X Toolkit functions can be called by the application, it must initialize the toolkit.

To initialize the X Toolkit, the application must call the `XtInitialize` function.

```
Widget XtInitialize(shell_name, application_class, options, num_options, argc, argv)
      String shell_name;
      String application_class;
      XrmOptionDescRec options[];
      Cardinal num_options;
      Cardinal *argc;
      String argv[];
```

shell_name Specifies the name of the application shell widget instance, which usually is something generic like "main".

application_class Specifies the class name of this application, which usually is the generic name for all instances of this application. By convention, the class name is formed by reversing the case of the application's first two letters. For example, an application named "xterm" would have a class name of "XTerm".

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 *Xlib - C Language X Interface*.

num_options Specifies the number of entries in options list.

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

argv Specifies the command line parameters.

`XtInitialize` builds the resource database, parses the command line, opens the display, and initializes the X Toolkit. It returns a `TopLevelShell` widget to use as the parent of the application's root widget.

`XtInitialize` modifies argc and argv to contain just the parameters that were not a display, geometry, or resource specification. If the modified argc is not zero (0), most applications simply print out the modified argv along with a message about the allowable options.

An application can have multiple top-level widgets. The widget returned by `XtInitialize` has the WM_COMMAND property set for session managers. See "Shell Widgets" for more information.

`XtInitialize` saves the application name and class_name for qualifying all widget resource specifiers. On UNIX-based systems, the application name is the final component of argv[0]. (This can be modified from the command line by specifying the -name option.) The application name and class_name are used as the left-most components in all widget resource names for this application.

4.2 Loading the Resource Database

XtInitialize loads the application's resource database from three sources in the following order:

- Application-specific class resource file
- Server resource file
- User's environment resource file

The application-specific resource file name is constructed from the class name of the application and points to a site-specific resource file that usually is installed by the site manager when the application is installed. On UNIX-based systems, the application resource file is
/usr/lib/X11/app-defaults/*class*, where class is the application class name.

The server resource file is the contents of the X server's RESOURCE_MANAGER property, as returned by XOpenDisplay. If no such property exists for the display, the .Xdefaults file in the user's home directory, if it exists, is loaded in place of the server property.

The user's environment resource file name is constructed by using the value of the user's XENVIRONMENT variable for the full path of the file. If this environment variable does not exist, XtInitialize looks in the user's home directory for the .Xdefaults-host file, where host is the name of the user's host machine. If the resulting resource file exists, XtInitialize loads its contents into the resource database.

4.3 Parsing the Command Line

XtInitialize first parses the command line looking for the following options:

-display	Specifies the display name for XOpenDisplay.
-synchronize	Calls XSynchronize to put Xlib into synchronous mode.
-name	Sets the resource name prefix in place of argv[0].

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

```
typedef enum {
      XrmoptionNoArg,            /* Value is specified in OptionDescRec.value    */
      XrmoptionIsArg,            /* Value is the option string itself    */
      XrmoptionStickyArg,        /* Value is characters immediately following option */
      XrmoptionSepArg,           /* Value is next argument in argv    */
      XrmoptionSkipArg,          /* Ignore this option and the next argument in argv */
      XrmoptionSkipLine          /* Ignore this option and the rest of argv    */
} XrmOptionKind;

typedef struct {
      char *option;              /* Option name in argv    */
      char *specifier;           /* Resource name (sans application name)    */
      XrmOptionKind argKind;     /* Which style of option it is    */
      caddr_t value;             /* Value to provide if XrmoptionNoArg    */
} XrmOptionDescRec, *XrmOptionDescList;
```

The standard table contains the following entries:

Option string	Resource name	Argument Kind	Resource value
-background	background	SepArg	next argument
-bd	borderColor	SepArg	next argument
-bg	background	SepArg	next argument
-borderwidth	borderWidth	SepArg	next argument
-bordercolor	borderColor	SepArg	next argument
-bw	borderWidth	SepArg	next argument
-display	display	SepArg	next argument
-fg	foreground	SepArg	next argument
-fn	font	SepArg	next argument
-font	font	SepArg	next argument
-foreground	foreground	SepArg	next argument
-geometry	geometry	SepArg	next argument
-iconic	iconic	NoArg	on
-name	name	SepArg	next argument
-reverse	reverseVideo	NoArg	on [not implemented]
-rv	reverseVideo	NoArg	on [not implemented]
+rv	reverseVideo	NoArg	off [not implemented]
-synchronize	synchronize	NoArg	on
-title	title	SepArg	next argument
-xrm	next argument	ResArg	next argument

NOTE

1. Any unique abbreviation for an option name in the standard table or in the application table is accepted.
2. For backwards compatibility with older command line syntax, an X Toolkit installation (compile time) option allows the following arguments on the command line:

=	geometryIsArg	this argument
:	display IsArg	this argument

The colon (:) can appear anywhere within the argument, and the argument will be accepted as the display string, if the -display argument is not specified on the command line.

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

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

When it fully parses the command line, `XtInitialize` merges the application option table with the standard option table and then calls the Xlib `XrmParseCommand` function. An entry in the application table with the same name as an entry in the standard table over-rides the standard table entry. If an option name is a prefix of another option name, both names are kept in the merged table. Although option tables need not be sorted by option name, `XrmParseCommand` is somewhat more efficient if they are.

4.4 Obtaining Window Information from a Widget

The Core widget definition contains the screen and window IDs. The window field may be NULL for a while (see “Creating Widgets” and “Realizing Widgets”).

The display pointer, the parent widget, screen pointer, and window of a widget are returned by the following macros:

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

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

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

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

These macros take a widget and return the specified value.

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

The x, y, width, height, and border_width window attributes are available to geometry managers. These fields are maintained synchronously inside the X Toolkit. When an XConfigureWindow is issued on the widget's window (on request of its parent), these values are updated immediately rather than sometime later when the server gets around to generating a ConfigureNotify event. (In fact, most widgets do not have SubstructureNotify turned on.) This ensures that all geometry calculations are based on the internally consistent toolkit world, rather than on either of the following:

- An inconsistent world updated by asynchronous ConfigureNotify events
- A consistent but slow world in which geometry managers ask the server for window sizes whenever they need to layout their managed children. See "Geometry Management" for further information.

4.5 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 that, then, get mapped.

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

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

Notifying a parent about its managed set involves geometry layout and possibly geometry negotiation. A parent deals with constraints on its size imposed from above (as when a user specifies the application window size), and suggestions made from below (as when a primitive child computes its preferred size). The clash 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 do not really know where anything should go on the screen until this process settles down.

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

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

4.5.1 Creating and Merging Argument Lists

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

```
typedef long XtArgVal;

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

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

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

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

arg Specifies the name-value pair to set.

name Specifies the name of the resource.

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

An `ArgList` usually is specified in a highly stylized manner in order 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);
```

NOTE

You should not use auto-increment or auto-decrement within the first argument to `XtSetArg`. As it is currently implemented, `XtSetArg` is a macro that dereferences the first argument twice.

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

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

args1 Specifies the first `ArgList` to include.

num_args1 Specifies number of arguments in the first `ArgList`.

args2 Specifies the second ArgList to include.

num_args2 Specifies the number of arguments in the second `ArgList`.

`XtMergeArgLists` allocates storage large enough to hold the combined `ArgList` structures and copies them into it. It does not check for duplicate entries. When it is no longer needed, the returned storage should be freed by the client with `XtFree`.

4.5.2 Creating a Widget Instance

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

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

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

widget_class Specifies the widget class pointer for the created widget.

parent Specifies the parent widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in args. The number of arguments in an argument list can be automatically computed by using the `XtNumber` macro. For further information, see “Determining the Number of Elements”.

`XtCreateWidget` performs much of the “boiler-plate” operations of widget creation. That is, it performs the following:

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

4.5.3 Creating an Application Shell Instance

To create an instance of an application shell widget, use `XtCreateApplicationShell`.

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

name Specifies the resource name for the created application shell widget.

widget_class Specifies the widget class pointer for the created application shell widget. This will usually be `topLevelShellWidgetClass` or a subclass thereof.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in args.

`XtCreateApplicationShell` creates another top-level widget that is the root of another widget tree. The initial top-level widget is returned from `XtInitialize`. An application uses this procedure if it needs to have several independent windows.

4.5.4 Initialization of a Widget Instance

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

```
typedef void (*XtInitProc)();

void InitProc(request, new)
      Widget request, new;
```

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

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

The three main jobs of an initialization procedure are to:

- Allocate space for and copy any resources that are referenced by address. For example, if a widget has a field that is a string (char *) it cannot depend upon the characters at that address remaining constant but must dynamically allocate space for the string and copy it to the new space. (Note that you should not allocate space for or copy callback lists.)
- Compute values for unspecified resource fields. For example, if width and height are zero (0), the widget should compute a nice width and height based on other resources. This is the only time that a widget may ever directly assign its own width and height.
- Compute values for uninitialized non-resource fields that are derived from resource fields. For example, GCs that the widget uses are derived from resources like background, foreground, and font.

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

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

If a subclass does not need an initialization procedure because it does not need to perform any of the above operations, you 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. How does a subclass know the difference between a specified size and a size computed by a superclass?

The request and new parameters provide the necessary information. The "request" widget is the widget as originally requested. The "new" widget starts with the values in the request, but it has been updated by all superclass 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, if the initialization procedure needs to call any routines that operate on a widget, it should specify "new" as the widget instance.

4.5.5 Initialization of a Constraint Widget Instance

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

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

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

4.5.6 Initialization of Nonwidget Data

The initialize_hook procedure is of type `XtArgsProc`:

```
typedef void (*XtArgsProc)();

void ArgsProc(w, args, num_args)
     Widget w;
     ArgList args;
     Cardinal *num_args;
```

w Specifies the widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in args.

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

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

4.6 Realizing Widgets

To realize a widget instance, use `XtRealizeWidget`.

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

w Specifies the widget.

`XtRealizeWidget` performs the following:

- If the widget is already realized, `XtRealizeWidget` simply returns.
- Otherwise, it makes a post-order traversal of the widget tree rooted at the specified widget and calls the change_managed procedure of each composite widget that has one or more managed children.
- It then 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 a primitive widget, nothing else need be done, and `XtRealizeWidget` returns. Otherwise, it recursively descends to each of the widget's managed children and calls the realize procedures.

- Finally, `XtRealizeWidget` maps all of the managed children windows that have mapped_when_managed TRUE. (If a widget is managed, but mapped_when_managed is FALSE, the widget is allocated visual space but is not displayed. Some people seem to like this to indicate certain states.)

If num_children equals num_mapped_children, `XtRealizeWidget` calls `XMapSubwindows` to map all the children at once. Otherwise, it maps each child individually. If the widget is a top-level shell widget (that is, it has no parent), `XtRealizeWidget` maps the widget window.

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

- If w is realized, then all managed children of w are realized.

- If w is realized, then all managed children of w that are also mapped_when_managed are mapped.

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

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

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

w Specifies the widget.

`XtIsRealized` returns TRUE if the widget has been realized. That is, it returns TRUE if the widget has a nonzero X window ID.

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

4.6.1 Creation of a Window for a Widget Instance

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

```
typedef void (*XtRealizeProc)();

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

w Specifies the widget.

value_mask Specifies which fields in the attributes structure to use.

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

The realize procedure must make the window a reality.

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

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

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

A widget class can inherit its realize procedure from its superclass during class initialization. The realize procedure defined for Core simply calls `XtCreateWindow` with the passed value_mask and attributes, and with windowClass and visual set to `CopyFromParent`. Both `CompositeWidgetClass` and `ConstraintWidgetClass` inherit this realize procedure, and most new widget subclasses can do the same. See "Inheriting Superclass Operations" for further information.

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, Label sets bit_gravity to `WestGravity`, `CenterGravity`, or `EastGravity`. Consequently, shrinking a Label just moves the bits appropriately, and no `Expose` event is needed for repainting.

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

4.6.2 Create Window Convenience Routine

Rather than call the Xlib `XCreateWindow` function explicitly, a realize procedure should call the X Toolkit analog `XtCreateWindow`. This routine 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 used to set x, y, and so on

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

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

value_mask Specifies which fields in attributes to use.

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

`XtCreateWindow` calls `XCreateWindow` with values from the widget structure and the passed parameters. Then, it assigns the created window into the widget's window field.

`XtCreateWindow` evaluates the following fields of the Core widget structure:

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

4.7 Destroying Widgets

Destroying widgets is simple. The X Toolkit provides support to:

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

To destroy a widget instance, use XtDestroyWidget.

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

w   Specifies the widget.

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

In phase one, XtDestroyWidget performs the following actions:

- If the being_destroyed field of the widget is TRUE, XtDestroyWidget returns immediately.
- Removes the widget from its parent's managed set which, in turn, causes the widget to be unmapped.
- Sets the being_destroyed field to TRUE and the visible bit to FALSE for the widget and all descendants.
- 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:

- If w1 occurs before w2 on the destroy list, then there is no ancestor/child relationship between the two, or w1 is a descendant of w2. (The terms "child" and "descendant" here refer to both normal and pop-up children.)

Phase two occurs when all procedures that should execute as a result of the current event have been called (including all procedures registered with the Event and Translation Managers). That is, phase two occurs when `XtNextEvent` is called.

NOTE

1. `XtDestroyWidget` may get rid of all widgets, and then the next call to `XtNextEvent` won't ever get any events. So we expect to move phase two to happen at the end of `XtDispatchEvent`, allowing customized event loops to test a flag before looping back to `XtNextEvent`.
2. The phase two destroy should happen only at the end of the outermost call to `XtDispatchEvent`, because there may be nested calls to an event dispatch loop in applications/widgets that maintain some state in the program counter.

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

- Calls the destroy callbacks registered on the widget (and all descendants) in post-order. That is, it calls children callbacks before parent callbacks.
- Calls the widget's parent's delete_child procedure. (See "Deletion of Children".)
- If the widget's parent is a subclass of `constraintWidgetClass`, it calls the constraint destroy procedure for the parent, then the parent's superclass, until finally it calls the constraint destroy procedure for `constraintWidgetClass`.
- Calls the destroy class procedures for the widget (and all descendants) in post-order. For each such widget, it calls the destroy procedure declared in the widget class, then the destroy procedure declared in its superclass, until finally it calls the destroy procedure declared in the Core class record.
- Calls `XDestroyWindow` if the widget is realized (that is, has an X window). The server recursively destroys all descendant windows.

Finally, `XtDestroyWidget` recursively descends the tree and deallocates all widgets and constraint records.

Since XTDestroyWidget automatically destroys all children when a parent is destroyed, there is no need for applications to explicitly destroy the children. If an application destroys a widget that has any popup children and then destroys its children, a segmentation fault occurs.

4.7.1 Adding and Removing Destroy Callbacks

The destroy callback uses the mechanism described in "Callbacks". The destroy callback list is identified by the resource name XtNdestroyCallback. To add a destroy callback procedure ClientDestroy with client data *client_data* to Widget w, call XtAddCallback. To remove the callback, call XtRemoveCallback. Both calls take the following parameter list:

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

4.7.2 Deallocation of Dynamic Data

The destroy procedure is of type XtWidgetProc:

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

void WidgetProc(w)
        Widget w;
```

w Specifies the widget.

The destroy procedures are called in subclass-to-superclass order. Therefore, a widget's destroy procedure should only deallocate storage that is specific to the subclass and should not bother with the storage allocated by any of its superclasses. If a widget does not need to deallocate any storage, the destroy procedure entry in its widget class record can be NULL.

Deallocating storage includes but is not limited to:

- Calling XtFree on dynamic storage allocated with XtMalloc, XtCalloc, and so on.
- Calling XtRemoveAllCallbacks on callback lists.
- Calling XtDestroyPixmap on pixmaps allocated with XtGetPixmap.
- Calling XFreePixmap on pixmaps created with direct X calls.
- Calling XtDestroyGC on GCs allocated with XtGetGC.
- Calling XFreeGC on GCs allocated with direct X calls.

- Calling XtRemoveEventHandler on event handlers added with XtAddEventHandler.
- Calling XtRemoveTimeOut on timers created with XtAddTimeOut.

4.7.3 Deallocation of Dynamic Constraint Data

The constraint destroy procedure is of type XtWidgetProc. They are called for a widget whose parent is a subclass of constraintWidgetClass. The constraint destroy procedures are called in subclass-to-superclass order, starting at the widget's parent and ending at constraintWidgetClass. Therefore, a parent's constraint destroy procedure should only deallocate storage that is specific to the constraint subclass and not the storage allocated by any of its superclasses.

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

4.8 Exiting an Application

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

Callbacks 5

Applications and other widgets (clients) often want to register a procedure with a widget that gets called under certain conditions. For example, when a widget is destroyed, every procedure on the widget's destroy_callbacks list is called to notify clients of the widget's impending doom.

Every widget has a destroy_callbacks list. Widgets can define additional callback lists as they see fit. For example, the Command widget has a callback list to notify clients when the button has been activated.

5.1 Callback Procedure and Callback List Definitions

Callback procedures for use in callback lists are of type `XtCallbackProc`:

```
typedef void (*XtCallbackProc)();

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

w Specifies widget for which the callback is registered.

client_data Specifies the data that the widget should pass back to the client when the widget executes the client's callback procedure. This is a way for the client registering the callback to also register client-specific data: a pointer to additional information about the widget, a reason for invoking the callback, and so on. It is perfectly normal to have client_data be NULL if all necessary information is in the widget.

call_data Specifies any callback-specific data the widget wants to pass to the client. For example, when Scrollbar executes its thumbChanged callback list, it passes the new position of the thumb. The call_data argument merely is a convenience to avoid having simple cases where the client could otherwise need to call `XtGetValues` or a widget-specific function to retrieve data from the widget. Complex state information in call_data generally should be avoided. The client can use the more general data retrieval methods, if necessary.

Whenever a client wants to pass a callback list as an argument in an XtCreateWidget, XtSetValues, or XtGetValues call, it should specify the address of a NULL-terminated array of type XtCallbackList:

```
typedef struct {
      XtCallbackProc callback;
      caddr_t closure;
} XtCallbackRec, *XtCallbackList;
```

For example, the callback list for procedures A and B with client data clientDataA and clientDataB, respectively, is:

```
static XtCallbackRec callbacks[] = {
      {A, (caddr_t) clientDataA},
      {B, (caddr_t) clientDataB},
      {(XtCallbackProc) NULL, (caddr_t) NULL}
};
```

Though callback lists are passed by address in argument lists, the Intrinsics know about callback lists. Your widget initialize and set_values procedures should not allocate memory for the callback list. The Intrinsics do this for you, using a different structure for their internal representation.

5.2 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 using the exported resource name. All callback manipulation routines described below check that the requested callback list is indeed implemented by the widget.

In order for the Intrinsics to find and correctly handle callback lists, they should always be declared with a resource type of XtRCallback.

5.3 Adding Callback Procedures

To add a callback procedure to a callback list, use XtAddCallback.

```
void XtAddCallback(w, callback_name, callback, client_data)
      Widget w;
      String callback_name;
      XtCallbackProc callback;
      caddr_t client_data;
```

w Specifies the widget to add the callback to.

callback_name Specifies the callback list within the widget to append to.

callback Specifies the callback procedure to add.

client_data Specifies the client data to be passed to the callback when it is invoked by `XtCallCallbacks`. (The client_data parameter is often NULL).

To add a list of callback procedures to a callback list, use `XtAddCallbacks`.

```
void XtAddCallbacks(w, callback_name, callbacks)
      Widget w;
      String callback_name;
      XtCallbackList callbacks;
```

w Specifies the widget to add the callbacks to.

callback_name Specifies the callback list within the widget to append to.

callbacks Specifies the null-terminated list of callback procedures and corresponding client data to add.

5.4 Removing Callback Procedures

To remove a callback procedure from a callback list, use `XtRemoveCallback`.

```
void XtRemoveCallback(w, callback_name, callback, client_data)
      Widget w;
      String callback_name;
      XtCallbackProc callback;
      caddr_t client_data;
```

w Specifies the widget to delete the callback from.

callback_name Specifies the callback list within the widget to remove the callback from.

callback Specifies the callback procedure to delete.

client_data Specifies the client data to match on the registered callback procedure. (The `XtRemoveCallback` routine removes a callback only if both the procedure and the client data match).

To remove a list of callback procedures from a callback list, use `XtRemoveCallbacks`.

```
void XtRemoveCallbacks(w, callback_name, callbacks)
      Widget w;
      String callback_name;
      XtCallbackList callbacks;
```

w Specifies the widget to delete the callbacks from.

callback_name Specifies the callback list within the widget to remove the callbacks from.

callbacks Specifies the list of callbacks to delete.

To remove all callback procedures from a callback list (and, thus, free all storage associated with the callback list), use `XtRemoveAllCallbacks`.

```
void XtRemoveAllCallbacks(w, callback_name)
      Widget w;
      String callback_name;
```

w Specifies the widget to remove the callback from.

callback_name Specifies the callback list within the widget to remove.

5.5 Executing Callback Procedures

To execute the procedures in a callback list, use `XtCallCallbacks`.

```
void XtCallCallbacks(w, callback_name, call_data)
      Widget w;
      String callback_name;
      caddr_t call_data;
```

w Specifies the widget containing the callback list that is to be executed.

callback_name Specifies the callback list within the widget to execute.

call_data Specifies a callback-list specific data value to pass to each of the callback procedure in the list. The call_data is NULL if no data is needed (for example, the commandActivated callback list in Command needs only to notify its clients that the button has been activated). The call_data is the actual data if only one (32-bit) long word is needed. The call_data is the address of the data if more than one word is needed.

5.6 Checking the Status of a Callback List

To find out the status of a callback list, use `XtHasCallbacks`.

```
typedef enum {XtCallbackNoList, XtCallbackHasNone, XtCallbackHasSome} XtCallbackStatus;

XtCallbackStatus XtHasCallbacks(w, callback_name)
      Widget w;
      String callback_name;
```

w Specifies the widget to check.

callback_name Specifies the callback list within the widget to check.

`XtHasCallbacks` first checks if the widget has a callback list identified by callback_name. If not, it returns `XtCallbackNoList`. Otherwise, it returns `XtCallbackHasNone` if the callback list is empty, and `XtCallbackHasSome` if the callback list has at least one callback registered.

This page left blank intentionally.

Composite Widgets 6

Composite widgets can have children. Consequently, they are responsible for much more than primitive widgets. Their responsibilities (either implemented directly by the widget class or indirectly by Intrinsic procedures) 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.
- Focus management for the displayable 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 all children of a destroyed composite widget also get destroyed.

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

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

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 will not be mapped automatically if their map_when_managed field is FALSE. This field default is TRUE and is changed by using `XtSetMappedWhenManaged`.

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

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

Each composite widget, especially those that are heterogeneous, can define ways for one child to change focus to another child by means of the move_focus_to_next and move_focus_to_prev procedures. For example, typing carriage return in one child widget may move to the "next" child widget, while typing a number in one child widget may move focus to any of a number of children widgets.

6.1 Verifying the Class of a Composite Widget

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

```
void XtIsComposite(w)
      Widget w;
```

w Specifies the widget under question.

`XtIsComposite`(w) is just an abbreviation for `XtIsSubclass` (*w*, *compositeWidgetClass*).

6.2 Addition of Children to a Composite Widget

To add the child to the parent's children array, XtCreateWidget calls the parent's class routine insert_child. The insert_child procedure for a composite widget is of type XtWidgetProc. An insert_child procedure takes the widget to insert and the argument list used to create the widget.

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

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

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

6.2.1 Insertion Order of Children

Instances of composite widgets may care 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, while it may want buttons that represent file names to be kept in alphabetical order.

The insert_position procedure for a composite widget instance is of type XtOrderProc:

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

Cardinal OrderProc(w)
      Widget w;
```

w Specifies the widget.

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

The return value of the insert_position procedure indicates how many children should go before the widget. Returning zero (0) means before all other children, while returning num_children means after all other children. The default insert_position function returns num_children. This can be overridden by a specific composite widget's resource list or by the argument list provided when the composite widget is created.

6.3 Deletion of Children

XtDestroyWidget eventually causes a call to the parent's class routine delete_child in order to remove the child from the parent's children array.

A deletion procedure is of type XtWidgetProc, and it takes the widget being deleted.

Most widgets just inherit delete_children from their superclass. Composite widgets that create companion widgets define their own delete_children routine to remove these companion widgets.

6.4 Adding and Removing Children from the Managed Set

The X Toolkit provides a set of generic routines to permit the addition of widgets to or the removal of widgets from a composite widget's managed set. These generic routines eventually call the widget's class procedure, change_managed, which is of type XtWidgetProc.

6.4.1 Managing Children

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

```
typedef Widget *WidgetList;

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

children	Specifies a list of children to add.
num_children	Specifies the number of children to add.

XtManageChildren performs the following:

- Issues an error if the children do not all have the same parent.

- Returns immediately if the common parent is being destroyed.
- Otherwise, for each unique child on the list:
  - The child is ignored if it is already managed or being destroyed.
  - Otherwise, the child is marked as managed, and the parent's num_mapped_children field is incremented if the child has map_when_managed TRUE.
- If the parent is realized and after all children have been marked, `XtManageChildren` makes some of the newly managed children visible:
  - 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 only bother with 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 the that 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 the `XtMoveWidget` procedure. `XtMoveWidget` first updates the x and y fields and then calls `XMoveWindow` if the widget is realized.

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

To add a single child to the managed children of its parent widget, the application must first create the widget by using `XtCreateWidget` and then call `XtManageChild`.

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

child   Specifies the child to add.

XtManageChild constructs a WidgetList of length one (1) and calls XtManageChildren.

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

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

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

XtCreateManagedWidget is a convenience routine that calls XtCreateWidget followed by XtManageChild.

6.4.2 Unmanaging Children

To remove a list of children from the managed list of their parent, use XtUnmanageChildren.

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

children	Specifies the children to remove.
num_children	Specifies the number of children to remove.

XtUnmanageChildren performs the following:

- Issues an error if the children do not all have the same parent.
- Returns immediately if the common parent is being destroyed.
- Otherwise, for each unique child on the list:
    - The child is ignored if it is already unmanaged or being destroyed.
    - Otherwise, XtUnmanagedChildren marks the child as unmanaged.

    - If the child is realized, XtUnmanageChildren makes it non-visible by unmapping it.
    - Decrements the parent's num_mapped_children field if the widget has map_when_managed TRUE.
- Calls the change_managed routine of the widgets' parent after all children have been marked if the parent is realized.

XtUnmanageChildren does not destroy the children widgets. Removing widgets from a parent's managed set is often a temporary banishment, and, some time later, you may add the children again. To destroy widgets entirely, see "Destroying Widgets".

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

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

child Specifies the child to remove.

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

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

6.5 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 map_when_managed field, use XtSetMappedWhenManaged.

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

w Specifies the widget.

map_when_managed Specifies the new value (either True or False).

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

When a widget's mapped_when_managed field is False, the client is responsible for mapping and unmapping the widget.

`XtSetMappedWhenManaged` is a convenience function that is equivalent to (but slightly faster than) calling `XtSetValues` and setting the new value for the mappedWhenManaged resource.

To map a widget explicitly, use `XtMapWidget`.

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

w   Specifies the widget.

To unmap a widget explicitly, use `XtUnmapWidget`.

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

w   Specifies the widget.

6.6 Constrained Composite Widgets

Constraint widgets are a subclass of Composite widgets. The name comes from the fact that they manage the geometry of their children based upon constraints associated with each child. Constraints can be as simple as information such as the maximum width and height the parent will allow the child to occupy, or they can be more complicated information such as how other children should change if this child is moved or resized.

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.

In order to make it easy for widgets and the Intrinsics to keep track of the constraints a parent associated with each child, every widget has a constraints field. This field is the address of a parent-specific structure containing constraint information about the child. If a child's parent is not a subclass of `constraintWidgetClass`, then the child's constraints field is NULL.

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

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

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

A subclass of this widget that also wished to maintain a minimum size would define its constraint record thus:

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

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

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

XtCreateWidget uses the constraint_size field to allocate a constraint record when a child is created. The constraint_size field gives the number of bytes occupied by a constraint record.

XtCreateWidget 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 a the parent can compute constraint fields that are derived from constraint resources and can possible move or resize the child to conform to the given constraints.

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

XtDestroyWidget 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.

This page left blank intentionally.

Pop-up Widgets 7

There are three kinds of pop-ups:

- Modeless pop-ups
- Modal pop-ups
- Spring-loaded pop-ups

A modeless dialog box, an example of a modeless pop-up, is normally visible to the window manager and looks much like just another application from the user point of view. (The application itself is a special form of a modeless pop-up.)

A modal dialog box, an example of a modal pop-up, is not normally visible to the window manager, and, except for events that occur in the dialog box, it disables user-event processing by the application.

A menu, an example of a spring-loaded pop-up, is not visible to the window manager and, except for events that occur in the menu, disables user-event processing by all applications.

Modal pop-ups and spring-loaded pop-ups are really almost the same thing and should be coded as such. In fact, the same widget (for example, a ButtonBox or Menu) can be used both as a modal pop-up or 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, they can require different processing by the Intrinsics. Further, button up takes down a spring-loaded pop-up no matter where the button up occurs.

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

7.1 Pop-ups and the Widget/Window Hierarchy

The one thing all pop-ups have in common is that they break the widget/window hierarchy. Pop-ups windows are not geometry constrained by their parent widget. Instead, they are window children of “root”. This means pop-ups are created and attached to their widget parent differently than from normal widget children.

Because the X Toolkit does not support multiple inheritance, and because you can pop up a widget that belongs to any arbitrary widget (for example, Command, MenuBar, Text, and so on), the CorePart record includes the list of the widget's pop-up children. This means that a primitive widget can be a parent, but this is very different from being a composite widget parent. Think of it more like being a godfather. The 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 parent of a pop-up widget does not actively manage its pop-up children. In fact, it usually never notices them or operates upon them.

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

When traversal is enabled and a pop-up shell with either exclusive or non-exclusive keyboard grab is displayed, the currently traversed to widget will not have its traversal highlight removed.

7.2 Creating a Pop-up Shell

In order to pop up some arbitrary widget, it must be the only child of a pop-up widget "shell". This shell is responsible for communication with the X window manager on geometry requests and is responsible for proper handling of the bookkeeping associated with actual pop up and pop down. Pop-up shells never allow more than one child.

This shell is always referred to as the "pop-up shell". The single (normal) child is always referred to as the "pop-up child". Both taken together are referred to as the "pop-up".

To create a pop-up shell, use XtCreatePopupShell.

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

name Specifies the text name for the created shell widget.

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

parent Specifies the parent widget.

args Specifies the argument list to override the resource defaults.

num_args Specifies the number of arguments in args.

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

To use a spring-loaded pop-up, the pop-up shell must be created at application start-up so that the translation manager can find the shell by name. Otherwise, the pop-up shell can be created "on-the-fly" when the pop-up is actually 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.

7.3 Creating Pop-up Children

Once a pop-up shell is created, the single child of the pop-up shell can be created. The two styles for this are:

- Static
- Dynamic

At application start up, an application can create the child of the pop-up shell. This is appropriate for pop-up children that are composed of a fixed set of widgets, and 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 time. 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. This minimizes application startup time and allows the pop-up child to reconfigure itself each time it is popped up. In this case, the pop-up child creation routine should "poll" the application to find out if it should change the sensitivity of any of its subparts.

Pop-up child creation does not map the pop-up, even if you create the child and call `XtRealizeWidget` on the pop-up shell. All pop-up shells automatically perform an `XtManageChild` on their child within their insert_child procedure. There is no need for the creator of a pop-up child to call `XtManageChild`.

All shells have pop-up and pop-down callbacks. These 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. Programmers should be aware that excessive use of pop-up callbacks can make popping up occur more slowly.

7.4 Mapping a Pop-up Widget

Pop-ups can be popped up through several mechanisms:

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

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

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

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

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

popup_shell Specifies the widget shell to pop up.

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

`XtPopup` performs the following actions:

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

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

- Calls `XtRealizeWidget`(popup_shell)
- Calls `XMapWindow`(popup_shell)

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

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

w Specifies the widget executing the callback.

client_data Specifies the pop-up shell to pop up.

call_data Specifies the callback data. This parameter is not used by this procedure.

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

w Specifies the widget executing the callback.

client_data Specifies the pop-up shell to pop up.

call_data Specifies the callback data. This parameter is not used by this procedure.

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

w Specifies the widget executing the callback.

client_data Specifies the pop-up shell to pop up.

call_data Specifies the callback data. This parameter is not used by this procedure.

Each of these routines calls XtPopup with the shell specified by the client data parameter and grab_kind set as the name specifies. XtCallbackNone specifies XtGrabNone, and so on. Each then sets the widget that executed the callback list to be insensitive.

The use of these routines in callbacks is not required. In particular, callbacks that create pop-up shells dynamically or that must do more than desensitizing the button will have custom code.

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

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

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

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

If MenuPopup is invoked upon ButtonPress (possibly with modifiers), the translation manager pops up the shell with grab_kind XtExclusive and spring_loaded TRUE. If MenuPopup is invoked upon EnterWindow (possibly with modifiers), the translation manager pops up the shell with grab_kind XtNonexclusive and spring_loaded FALSE. Otherwise, the translation manager generates an error. When the widget is popped up, the following actions are performed:

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

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

- Calls XtRealizeWidget(popup_shell)
- Calls XMapWindow(popup_shell)

Note that these actions are the same as those for XtPopup.

MenuPopup tries to find the shell by looking up the widget tree starting at the parent of the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that parent, it pops up the shell with the appropriate parameters. Otherwise, it moves up the parent chain as needed. If MenuPopup gets to the application widget and cannot find a matching shell, it generates an error.

7.5 Unmapping a Pop-up Widget

Pop-ups can be popped down through several mechanism:

- A call to XtPopdown.
- The supplied callback procedure XtCallbackPopdown.
- The standard translation action MenuPopdown.

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

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

popup_shell Specifies the widget shell to pop down.

XtPopdown performs the following:

- Calls XtCheckSubclass to ensure popup_shell is a subclass of Shell.
- Checks that popup_shell is currently popped_up. If not, it generates an error.
- Unmaps popup_shell's window.
- If popup_shell's grab_kind is either XtGrabNonexclusive or XtGrabExclusive, calls XtRemoveGrab.
- Sets popup_shell's popped_up field to FALSE.
- Calls the callback procedures on the shell's popdown_callback list.

Pop-ups that have been popped up with one of the callback routines (that is, XtCallbackNone, XtCallbackNonexclusive, XtCallbackExclusive) can be popped down by the callback routine XtCallbackPopdown.

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

w Specifies the widget executing the callback.

client_data Specifies the pop-up shell to pop down and the widget used to originally pop it up.

call_data Specifies the callback data. This parameter is not used by this procedure.

XtCallbackPopdown casts the client data parameter to an XtPopdownID pointer:

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

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

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

```
void MenuPopdown()
```

MenuPopdown calls XtPopdown with the widget for which the translation is specified.

Shell Widgets 8

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. That is, while clients have to create them, 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 will resize its child widget automatically. Similarly, if the shell's child widget wants to change size, it can make a geometry request to the shell, and the shell will negotiate the size change with the outer environment. Clients should never attempt to change the size of their shells directly.

There are three classes of public shells:

`OverrideShell` This class is used for shell windows that completely bypass the window manager. Pop-up menu shells will typically be of this class or a subclass.

`TransientShell` This class is used for shell windows that can be manipulated by the window manager but are not allowed to be separately iconified. They get iconified by the window manager if the main application shell is iconified. Dialog boxes that make no sense without their associated application will typically be in a shell of this class or a subclass.

`TopLevelShell` This class is used for normal top level windows. Any additional top-level widgets an application needs will typically be in a shell of this class or a subclass.

8.1 Shell Widget Definitions

Widgets negotiate their size and position with their parent widget, the widget that directly contains them. Widgets at the top level of the hierarchy do not have parent widgets; instead they must deal with the outside world. To provide for this, top level widgets are encapsulated in special widgets called "Shells".

Shell widgets are subclasses of the Composite widget. They encapsulate other widgets and can allow a widget to "jump out" of the geometry clipping imposed by the parent/child window relationship. If desired, they provide a layer of communication with the window manager.

There are six different types of shell:

`Shell` This is the base class for shell widgets and provides fields needed for all types of shells. `Shell` is a direct subclass of `Composite`.

`OverrideShell` This class is used for shell windows that completely bypass the window manager. It is a subclass of `Shell`.

`WMShell` Contains fields needed by the common window manager protocol. It is a subclass of `Shell`.

`VendorShell` Contains fields used by vendor-specific window managers. It is a subclass of `WMShell`.

`TransientShell` This class is used for shell windows that can be manipulated by the window manager but are not allowed to be iconified. It is a subclass of `VendorShell`.

`TopLevelShell` This class is used for normal top level windows. It is a subclass of `VendorShell`.

The classes `Shell`, `WMShell`, and `VendorShell` are internal and should not be instantiated. Only `OverrrideShell`, `TransientShell`, and `TopLevelShell`. are for public use.

If a shell is to match its child size automatically, the child widget must be created before the shell is realized.

8.1.1 ShellClassPart Definitions

No shell widget classes have any additional fields:

```
typedef struct { int empty; } ShellClassPart, OverrideShellClassPart,
      WMShellClassPart, VendorShellClassPart, TransientShellClassPart,
      TopLevelShellClassPart;
```

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

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

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


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


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


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


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

The predefined class records and pointers for shells are:

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

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

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

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

8.1.2 ShellPart Definition

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

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


typedef struct { int empty; } OverrideShellPart;


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

typedef struct {
      int vendor_specific;
} VendorShellPart;


typedef struct { int empty; } TransientShellPart;

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

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

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

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

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

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


typedef struct {
      CorePart core;
      CompositePart composite;
      ShellPart shell;
      WMShellPart wm;
      VendorShellPart vendor;
      TransientShellPart transient;
} TransientShellRec, *TransientShellWidget;


typedef struct {
      CorePart core;
      CompositePart composite;
      ShellPart shell;
      WMShellPart wm;
      VendorShellPart vendor;
      TopLevelShellPart topLevel;
} TopLevelShellRec, *TopLevelShellWidget;
```

8.1.3 ShellPart Default Values

The default values for all shell fields (filled in by the Shell resource lists and the Shell initialize procedures) are:

Field	Default Value
geometry	NULL
create_popup_child_proc	NULL
grab_kind	(internal)
spring_loaded	(internal)
popped_up	(internal)
allow_shell_resize	FALSE
client_specified	(internal)
save_under	FALSE
override_redirect	TRUE for `OverrideShells`, FALSE otherwise
popup_callback	NULL
popdown_callback	NULL
title	Icon name, if specified, otherwise the application's name
wm_timeout	5 seconds
wait_for_wm	TRUE
transient	TRUE for `TransientShells`, FALSE otherwise

The geometry resource can be used to specify size and position. This is usually done only from a command line or a defaults file. The create_popup_child_proc is called by the `XtPopup` procedure and usually is NULL. The allow_shell_resize field controls whether or not the widget contained by the shell is allowed to try to resize itself. If it is FALSE, any geometry requests will always return `XtGeometryNo`. Setting save_under instructs the server to attempt to save the contents of windows obscured by the shell when it is mapped and to restore it automatically later. It is useful for pop-up menus. Setting overrideRedirect determines whether or not the shell window will be visible to the window manager. If TRUE, the window will be immediately mapped without the manager's intervention. The popup and popdown callbacks are called during `XtPopup` and `XtPopdown`. The title is a string to be displayed by the window manager. The wm_timeout resource limits the amount of time a shell will wait for confirmation of a geometry request to the window manager. If none comes back within that time, the shell decides the window manager is broken and sets wait_for_wm to be FALSE (Later events may reset this value). The wait_for_wm resource sets the initial state for this flag. When the flag is FALSE, the shell does not wait for a response. Rather, it relies upon asynchronous notification. All other resources are for fields in the window manager hints and the window manager size hints. For further information, see *Xlib - C Language X Interface*.

`Transient` and `TopLevel` shells all have the following extra resources:

Resource	Default Value
minWidth	none
minHeight	none
maxWidth	none
maxHeight	none
widthInc	none
heightInc	none
minAspectX	none
minAspectY	none
maxAspectX	none
maxAspectY	none
input	FALSE
initialState	Normal
iconPixmap	none
iconWindow	none
iconX	none
iconY	none
IconMask	none
windowGroup	none

TopLevel shells have the the following additional resources:

Field	Default Value
icon_name	Shell widget's name
iconic	FALSE

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

Utility Functions 9

The X Toolkit provides a number of utility functions for:

- Memory management
- Sharing graphics contexts
- Exposure regions
- Error handling

9.1 Memory Management

The X Toolkit memory management routines provide uniform checking for null pointers, and error reporting on memory allocation errors. These routines are completely compatible with the standard C language runtime routines malloc, calloc, realloc, and free with the added functionality:

- XtMalloc, XtNew, 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 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.

XtMalloc 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 XtError.

To allocate storage for a new instance of a data type, use XtNew.

```
type *XtNew(type);
      type;
```

type Specifies a previously declared data type

XtNew returns a pointer to the allocated storage. If there is insufficient memory to allocate the new block, XtNew calls XtError. XtNew is an abbreviation for:

```
      ((type *) XtMalloc((unsigned) sizeof(type))
```

To allocate and initialize an array, use XtCalloc.

```
char *XtCalloc(num, size);
      Cardinal num;
      Cardinal size;
```

num Specifies the number of array elements to allocate.

size Specifies the size of an array element in bytes.

XtCalloc allocates space for the specified number of array elements of the specified size bytes and initializes the space to zero. If there is insufficient memory to allocate the new block, XtCalloc calls XtError.

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 old storage.

num Specifies number of bytes desired in new storage.

XtRealloc 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 XtError. If the specified ptr argument is NULL, XtRealloc allocates the new storage without copying the "old" contents. That is, it simply calls XtMalloc.

To free an allocated block of storage, use XtFree.

```
void XtFree(ptr);
      char *ptr;
```

ptr Specifies a pointer to the block of storage that is to be freed.

XtFree returns storage and allows it to be reused. If the specified ptr argument is NULL, XtFree returns immediately.

9.2 Sharing Graphics Contexts

The X Toolkit provides a mechanism whereby cooperating clients can share Graphics Contexts, thereby, reducing both the number of Graphics Contexts created and the total number of server calls in any given application. The mechanism implemented is a simple caching scheme and all Graphics Contexts obtained by means of this mechanism must be treated as read-only. If a changeable Graphics Context is needed, the XCreateGC Xlib function should be used instead.

To obtain shared GCs, use XtGetGC.

```
GC XtGetGC(w, value_mask, values)
      Widget w;
      XtGCMask value_mask;
      XGCValues *values;
```

w Specifies the widget.

value_mask Specifies which fields of the values are specified. (See XCreateGC.)

values Specifies the actual values for this GC. (See XCreateGC.)

XtGetGC returns a Graphics Context. The parameters to this function are the same as those for XCreateGC, except that a widget is passed instead of a Display.

XtGetGC shares only GCs in which all values in the GC returned by XCreateGC are the same. In particular, it does not use the value_mask provided to determine which fields of the GC a widget considers relevant. The value_mask is used only to tell the server which fields should be filled in with widget data and which it should fill in with default values.

To deallocate a graphics context when it is no longer needed, use XtDestroyGC.

```
void XtDestroyGC(gc)
      GC gc;
```

gc Specifies the gc to be deallocated.

References to sharable GCs are counted and a free request is generated to the server when the last user of a GC destroys it.

9.3 Merging Exposure Events into a Region

The X Toolkit provides the `XtAddExposureToRegion` utility function that merges `Expose` and `GraphicsExpose` events into a region that clients can process at once, rather than processing individual rectangles. (For further information about regions, see *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`).

`XtAddExposureToRegion` 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 "Exposure Compression").

9.4 Translating Strings to Widget Instances

To translate a widget name to widget instance, use `XtNameToWidget`.

```
Widget XtNameToWidget(reference, names);
      Widget reference;
      String names;
```

reference Specifies the widget to start searching from.

names Specifies the fully qualified name of the desired widget.

The names argument contains the name of a widget with respect to the reference widget parameter. The names argument can contain more than one widget name for widgets that are not direct children of the reference widget. A dot (".") separates each component name.

XtNameToWidget looks for a widget whose name is the first component in the names parameter and who is a child (pop-up or normal) of the reference widget. It then uses that widget as the new reference and repeats the search after deleting the first component from the specified names argument. XtNameToWidget returns NULL if it cannot find the specified widget.

If more than one child of the reference widget matches the name, XtNameToWidget may return any of the children. The X Toolkit does not require that all children of a widget have unique names. If the names argument contains more than one component and if more than one child matches the first component, XtNameToWidget may return NULL if the single branch that it follows does not contained the named widget. That is, XtNameToWidget does not back up and follow other matching branches of the widget tree.

9.5 Translating Widget Coordinates

To translate an x-y coordinate pair from widget coordinates to root coordinates, use XtTranslateCoords.

```
void XtTranslateCoords(w, x, y, rootx_return, rooty_return)
      Widget w;
      Position x, y;
      Position *rootx_return, *rooty_return;
```

w	Specifies the widget.
x *y*	Specify the widget-relative coordinates.
rootx_return *rooty_return*	Returns the root-relative x and y coordinates.

While XtTranslateCoords is similar to XTranslateCoordinates, it does not generate a server request because all the required information already is in the widget's data structures.

The Xlib function XtTranslateCoordinates should be used when running a reparenting window manager, since XtTranslateCoordinates may return incorrect position information in that case.

9.6 Translating a Window to a Widget

To translate a window and display pointer into a widget instance, use `XtWindowToWidget`.

```
Widget XtWindowToWidget(display, window)
      Display *display;
      Window window;
```

display Specifies the display on which the window is defined.

window Specify the window for which you want the widget.

9.7 Handling Errors

The X Toolkit allows a client to register a procedure to be called whenever a fatal or nonfatal error occurs. This facility is intended for error reporting and logging but not for error correction or recovery.

Error and warning handlers are of type `XtErrorHandler`:

```
typedef void (*XtErrorHandler)();

void ErrorHandler(message)
    String message;
```

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. Fatal error handlers should not return.

The default error handler provided by the X Toolkit is `_XtError`. On UNIX-based systems, it prints the message to standard error and terminates the application.

To call the installed fatal error procedure, use `XtError`.

```
void XtError(message)
      String message;
```

message Specifies the error message to report.

To register a procedure to be called on non-fatal error conditions, use `XtSetWarningHandler`.

```
void XtSetWarningHandler(handler)
      XtErrorHandler handler;
```

handler Specifies the new non-fatal error procedure. Warning handlers usually return.

The default warning handler provided by the X Toolkit is `_XtWarning`. On UNIX-based systems, it prints the message to standard error and returns to the caller.

To call the installed non-fatal error procedure, use `XtWarning`.

```
void XtWarning(message)
      String message;
```

message Specifies the non-fatal error message to report.

This page left blank intentionally.

Event Handling 10

While X 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 merely 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 (see `XtMainLoop`, which reads events and dispatches them by calling the procedures that widgets have registered.

The event manager is a collection of routines to:

- Add or remove event sources other that 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.
- Focus keyboard events within a composite widget to a particular child.
- Call the appropriate set of procedures currently registered when an event is read.

Most widgets do not need to call any of the event manager routines 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 `XtMainLoop`.

10.1 Adding and Deleting Additional Event Sources

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

The next two functions provide input gathering from files. The application registers the files with the X Toolkit read routine. When input is pending on one of the files, the registered callback procedures are invoked.

10.1.1 Adding and Removing Input Sources

To register a new file for input, use XtAddInput.

```
XtInputId XtAddInput(source, condition, proc, client_data)
      int source;
      caddr_t condition;
      XtInputCallbackProc proc;
      caddr_t client_data;
```

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

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

proc Specifies the procedure that is called when input is available.

client_data Specifies the parameter to be passed to proc when input is available.

The XtAddInput function registers with the X Toolkit read routine 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, the callback procedure is called.

Callback procedures that are called when there are file events are of type XtInputCallbackProc:

```
typedef void (*XtInputCallbackProc)();

void InputCallbackProc(client_data, source, id)
        caddr_t client_data;
        int source;
        XtInputId id;
```

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

source Specifies the source file descriptor generating the event.

id Specifies the ID returned from the corresponding XtAddInput call.

To discontinue a source of input, use XtRemoveInput.

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

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

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

10.1.2 Adding and Removing Timeouts

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

To create a timeout value, use `XtAddTimeOut`.

```
XtIntervalId XtAddTimeOut(interval, proc, client_data)
      unsigned long interval;
      XtTimerCallbackProc proc;
      caddr_t client_data;
```

interval Specifies the time interval in milliseconds.

proc Specifies the procedure to be called when time expires.

client_data Specifies the parameter to be passed to proc when it is called.

The `XtAddTimeOut` function creates a timeout and returns an identifier for it. The timeout value is set to interval. The callback procedure will be called when the time interval elapses, after which the timeout is removed.

Callback procedures that are called when timeouts expire are of type `XtTimerCallbackProc`:

```
typedef void (*XtTimerCallbackProc)();

void TimerCallbackProc(client_data, id)
        caddr_t client_data;
        XtIntervalId *id;
```

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

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

To clear a timeout value, use `XtRemoveTimeOut`.

```
void XtRemoveTimeOut(timer)
      XtIntervalId timer;
```

timer Specifies the unique identifier for the timeout request to be destroyed.

XtRemoveTimeOut removes the timeout. Note that timeouts are automatically removed once they trigger.

10.2 Filtering X Events

The event manager provides two filters that can be applied to X user events. These filters screen out events that are redundant or that are temporarily unwanted.

10.2.1 Pointer Motion Compression

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

10.2.2 Enter/Leave Compression

To throw out pairs of enter and leave events that have no intervening events, the widget class field compress_enter/leave should be TRUE. These enter and leave events will never be delivered to the client.

10.2.3 Exposure Compression

Many widgets will 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, while widgets with simple displays might ignore the region entirely and redisplay their whole window or might get the bounding box from the region and redisplay only that rectangle.

In either case, these widgets do not care about getting partial expose events. If the compress_exposure field in the widget class structure is TRUE, the Event Manager calls the widget's expose procedure only once for each series of exposure events. In this case, all Expose events are accumulated into a region. When the Expose event with count zero is received, the Event Manager replaces the rectangle in the event with the bounding box for the region and calls the widget's expose procedure passing the (modified) exposure event and the region.

If compress_exposure is FALSE, the Event Manager will call the widget's expose procedure for every exposure event, passing the event and a region argument of NULL.

10.2.4 Setting and Checking the Sensitivity State of a Widget

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

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

w Specifies the widget.

sensitive Specifies whether or not the widget should receive keyboard and pointer events.

Many widgets, especially those with callback lists that get executed in response to some user-initiated action (for example, clicking down or up), have a mode in which they take on a different appearance (for example, greyed out or stippled) and do not respond to user events.

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

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

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

- If parent has either sensitive or ancestor_sensitive FALSE, then all children have ancestor_sensitive FALSE.

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

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

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

w Specifies the widget that is to be checked.

To indicate whether or not user input events are being dispatched, `XtIsSensitive` returns TRUE or FALSE. If both core.sensitive and core.ancestor_sensitive are TRUE, `XtIsSensitive` returns TRUE. Otherwise, it returns FALSE.

10.3 Adding and Removing X Event Handlers

Event handlers are procedures that are called when specified events occurs in a widget. Most widgets will not need to use event handlers explicitly. Instead, they use the Intrinsic's translation manager. Event handlers are of the type `XtEventHandler`:

```
typedef void (*XtEventHandler)();

void EventHandler(w, client_data, event)
      Widget w;
      caddr_t client_data;
      XEvent *event;
```

w Specifies the widget that this event handler was registered with.

client_data Specifies the client specific information registered with the event handler. This is usually NULL if the event handler is registered by the widget itself.

event Specifies the triggering event.

To register an event handler procedure with the dispatch mechanism, use `XtAddEventHandler`.

```
void XtAddEventHandler(w, event_mask, nonmaskable, proc, client_data)
      Widget w;
      XtEventMask event_mask;
      Boolean nonmaskable;
      XtEventHandler proc;
      caddr_t client_data;
```

w Specifies the widget for which this event handler is being registered.

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

nonmaskable Specifies whether this procedure should be called on the nonmaskable events. These are event of type `GraphicsExpose`, `NoExpose`, `SelectionClear`, `SelectionRequest`, `SelectionNotify`, `ClientMessage`, and `MappingNotify`.

proc Specifies the client event handler procedure.

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

The `XtAddEventHandler` function registers a procedure with the dispatch mechanism that is to be called when an event that matches the mask occurs on the specified widget. If the procedure is already registered, the specified mask is ORed into the existing mask. If the widget is realized, `XtAddEventHandler` calls `XSelectInput`, if necessary.

To remove a previously registered event handler, use `XtRemoveEventHandler`.

```
void XtRemoveEventHandler(w, event_mask, nonmaskable, proc, client_data)
      Widget w;
      XtEventMask event_mask;
      Boolean nonmaskable;
      XtEventHandler proc;
      caddr_t client_data;
```

w Specifies the widget for which this procedure is registered.

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

nonmaskable Specifies the events for which to unregister this procedure.

proc Specifies the event handler procedure registered.

client_data Specifies the client data registered.

`XtRemoveEventHandler` stops the specified procedure from receiving the specified events. If the widget is realized , `XtRemoveEventHandler` calls `XSelectInput`, if necessary.

To stop a procedure from receiving any events (which will remove it from the widget's event_table entirely), call `XtRemoveEventHandler` with an event_mask of `XtAllEvents` and with nonmaskable TRUE.

10.3.1 Adding and Removing Event Handlers without Selecting Events

On occasions, clients need to register an event handler procedure with the dispatch mechanism without causing the server to select for that event. To do this, use `XtAddRawEventHandler`.

```
void XtAddRawEventHandler(w, event_mask, nonmaskable, proc, client_data)
      Widget w;
      XtEventMask event_mask;
      Boolean nonmaskable;
      XtEventHandler proc;
      caddr_t client_data;
```

w Specifies the widget for which this event handler is being registered.

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

nonmaskable Specifies whether this procedure should be called on the nonmaskable events.

proc Specifies the client event handler procedure.

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

This function has the same behavior as `XtAddEventHandler`, except that it does not affect the widget's mask and it never causes an `XSelectInput` for its events. Note that the widget might already have those mask bits set because of other non-raw, event handlers registered on it.

To remove a previously registered raw event handler, use `XtRemoveRawEventHandler`.

```
void XtRemoveRawEventHandler(w, event_mask, nonmaskable, proc, client_data)
      Widget w;
      XtEventMask event_mask;
      Boolean nonmaskable;
      XtEventHandler proc;
      caddr_t client_data;
```

w Specifies the widget for which this procedure is registered.

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

nonmaskable Specifies the events for which to unregister this procedure.

proc Specifies the event handler procedure registered.

client_data Specifies the client data registered.

`XtRemoveRawEventHandler` stops the specified procedure from receiving the specified events. Because the procedure is a raw event handler, this will not affect the widget's mask and will never cause a call on `XSelectInput`.

10.4 Constraining Events to a Cascade of Widgets

Some widgets lock out any user input to the application except input to that widget. These are called "modal" widgets.

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

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

Display-related events should be delivered outside the modal cascade so that expose events and the like keep the application's display up-to-date. Any event that occurs within the cascade is delivered normally. Events that are delivered to the most recent spring-loaded shell in the cascade if they occur outside the cascade are called "remap" events and consist of the following events: KeyPress, KeyRelease, ButtonPress, and ButtonRelease.

Events that are ignored if they occur outside the cascade are: MotionNotify, EnterNotify, and LeaveNotify. All other events are delivered normally.

XtPopup uses the procedures XtAddGrab and XtRemoveGrab to constrain user events to a modal cascade and subsequently to remove a grab when the modal widget goes away. There is usually no need to call them explicitly.

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

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

w Specifies the widget to add to the modal cascade.

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

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

XtAddGrab appends the widget (and associated parameters) to the modal cascade. XtAddGrab checks that exclusive is TRUE if spring_loaded is TRUE. If not, it generates an error.

When XtDispatchEvent tries to dispatch a user event when at least one modal widget is in the widget cascade, it 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 is 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 wish to restrict user input to the most deeply nested dialog box add a subdialog widget to the cascade with exclusive TRUE.

User events that occur within the active subset are delivered to the appropriate widget, which is usually a child or further descendant of the modal widget.

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

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

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

w Specifies the widget to remove from the modal cascade.

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

10.5 Focusing Events on a Child

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

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

w Specifies the widget for which the keyboard focus is to be set.

descendant Specifies the widget to which the keyboard event is to be sent or None.

If a future keyboard event (KeyPress or KeyRelease) occurs on the specified widget, XtSetKeyboardFocus causes XtDispatchEvent to remap and send the event to the specified descendant widget.

If widget A is an ancestor of widget W and if no modal cascade has been created, a keyboard event is defined as occurring within W if one of the following focus conditions are true:

- W has the X server input focus.
- The event occurs within A or a descendent of A, and W has the keyboard focus for A.
- The event occurs within A, and no descendant of A has the keyboard focus for A, and the pointer is within W.

If widget A is an ancestor of widget W, and if a modal cascade exists, a keyboard event is defined as occurring within W if A is in the active subset of the modal cascade and if one of the previous focus conditions are true.

When W acquires the X input focus or the keyboard focus for one of its ancestors, a FocusIn event is generated for descendant if FocusNotify events have been selected by the descendant. Similarly, when W looses the X input focus or the keyboard focus for one of its ancestors, a FocusOut event is generated for descendant if FocusNotify events have been selected by the descendant.

Widgets that want the input focus may call XSetInputFocus explicitly. To allow outside agents to cause a widget to get the input focus, every widget exports an accept_focus procedure. Widgets interested in knowing when they lose the input focus must use the Xlib focus notification mechanism explicitly, typically by specifying translations for FocusIn and FocusOut events. Widgets that never want the input focus should set their accept_focus procedure to NULL.

Composite widgets have two additional functions:

- move_focus_to_next
- move_focus_to_prev

These procedures (which may be NULL) move the focus to the next child widget that wants it and to the previous child widget that wants it, respectively. The definition of next and previous is left to each individual widget. In addition, composite widgets are free to implement other procedures to move the focus between their children. Both move_focus_to_next and move_focus_to_prev should be entered in the widget class action table, so that they are available to translation specifications.

10.6 Querying Event Sources

The event manager provides several routines to examine and read events (including file and timer events) that are in the queue.

The next three functions handle X Toolkit equivalents of the XPending, XPeekEvent, and XNextEvent Xlib calls.

To determine if there are any events on the input queue, use XtPending.

```
Boolean XtPending()
```

The XtPending returns a nonzero value if there are events pending from the X server or other input sources. If there are no events pending, it flushes the output buffer and returns a zero value.

To return the value from the head of the input queue without removing input from the queue, use XtPeekEvent.

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

To return the value from the head of the input queue, use XtNextEvent.

```
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, XtNextEvent flushes the X output buffer and waits for an event while looking at the other input sources and timeout values and calling any callback procedures triggered by them.

10.7 Dispatching Events

The X Toolkit provides functions that dispatch events to widgets or other application code. Every client interested in events on a widget uses `XtAddEventHandler` to register which events it is interested in and a procedure (event handler) that is to be called when the event happens in that window.

When an event is received, it is passed to a dispatching procedure. This procedure calls the appropriate event handlers and passes them the widget, the event, and client-specific data registered with each procedure. If there are no handlers for that event registered, the event is ignored and the dispatcher simply returns.

The order in which the handlers are called is not defined.

To send events to registered functions and widgets, use `XtDispatchEvent.` Usually, this procedure is not called by client applications (see `XtMainLoop`).

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

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

The `XtDispatchEvent` function sends those events to those event handler functions that have been previously registered with the dispatch routine. The most common use of `XtDispatchEvent` is to dispatch events acquired with the `XtNextEvent` or `XtPeekEvent` procedure. However, it also can be used to dispatch user-constructed events. `XtDispatchEvent` also is responsible for processing grabs and keyboard focus.

10.8 Processing Input

To process input, an application can call `XtMainLoop.`

```
void XtMainLoop()
```

`XtMainLoop` first reads the next incoming file, timer, or X event by calling `XtNextEvent.` Then, it dispatches this to the appropriate registered procedure by calling `XtDispatchEvent.` This is 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.

There is nothing special about XtMainLoop. It is simply an infinite loop that calls XtNextEvent 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 0 before circling back to the call to XtNextEvent.

10.9 Widget Exposure and Visibility

Every primitive widget and some composite widgets display data on the screen by means of raw X 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 re-exposed.

10.9.1 Redisplay of a Widget

The expose procedure for a widget class is of type XtExposeProc:

```
typedef void (*XtExposeProc)();
```

```
void ExposeProc(w, event, 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.

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

If the widget's compress_exposure field is FALSE (see "Exposure Compression"), region will always be NULL. If the widget's compress_exposure field is TRUE, event will contain the bounding box for region.

A small simple widget (for example, Label) can ignore the bounding box information in the event and just 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 will use the region as a clip list in a GC and ignore the event information.

The expose procedure is responsible for exposure of all superclass data as well as its own, This is because, in general, this operation cannot be cleanly broken up.

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

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

Label would have invert and highlight always FALSE and highlight_width zero(0). Command would dynamically set highlight and highlight_width, but it would leave invert always FALSE. Finally, Toggle would dynamically set all three.

In this case, the expose procedures for Command and Toggle inherit their superclass's expose procedure. For further information, see "Inheriting Superclass Operations".

10.9.2 Widget Visibility

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

The visible field in the Core widget structure provides a hint to the widget that it need not display data. This field is guaranteed TRUE (by the time an Expose event is processed) if the widget is visible and is usually FALSE if the widget is not visible.

Widgets can use or ignore the visible hint as they wish. If they ignore it, they should have visible_interest in their widget class record set FALSE. In such cases, the visible field is initialized TRUE and never changes. If visible_interest is TRUE, the Event Manager asks for VisibilityNotify events for the widget and updates the visible field accordingly.

10.10 Geometry Management - Sizing and Positioning Widgets

A widget does not directly control its size and location, which is the responsibility of the parent of that widget. 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 X Toolkit provides the Geometry Management mechanism. Almost all composite widgets have a geometry manager (geometry_manager field in the widget class record) that is responsible for the size, position, and stacking order of the widget's children. The only exception are fixed boxes, which create their children themselves and can ensure that their children will never make a geometry request.

Widgets that wish to change their size, position, border width, or stacking depth must not use X calls directly. Instead, they must ask their parent's geometry manager to make the desired changes. When a child makes a request of the parent's geometry manager, the geometry manager can do one of the following:

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

Geometry requests are always made by the child itself. Clients that wish to change the geometry of a widget should call `XtSetValues` on the appropriate geometry fields. Parents that wish to change the geometry of a child can use `XtMoveWidget` or `XtResizeWidget` at any time.

When the geometry manager is asked to change the geometry of a child, the geometry manager may also rearrange and resize any or all of the other children that it controls. The geometry manager can move children around freely using `XtMoveWidget`. When it resizes a child (that is, changes width, height, or border_width) other than the one making the request, it should do so by calling `XtResizeWidget`.

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

Because such cascaded arbitration of widget geometry can involve extended negotiation, windows are not actually allocated to widgets at application startup until all widgets are satisfied with their geometry. See "Realizing Widgets" and "Creating Widgets" for more details.

10.10.1 Making General Geometry Manager Requests

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

```
XtGeometryResult XtMakeGeometryRequest(w, request, reply_return)
      Widget w;
      XtWidgetGeometry *request;
      XtWidgetGeometry *reply_return;
```

w Specifies the widget ID of the widget that is making the request.

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

reply_return Returns the allowed widget size. If a widget is not interested in handling `XtGeometryAlmost`, the reply parameter can be NULL.

The return codes from geometry managers are:

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

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

```
typedef unsigned long XtGeometryMask;

typedef struct {
      XtGeometryMask request_mode;
      Position x, y;
      Dimension width, height;
      Dimension border_width;
      Widget sibling;
      int stack_mode;
} XtWidgetGeometry;
```

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

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

XtMakeGeometryRequest, in exactly the same manner as the Xlib routine XConfigureWindow, uses the request_mode to determine which fields in the XtWidgetGeometry structure you want to specify.

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

```
#define Above          0
#define Below          1
#define TopIf          2
#define BottomIf       3
#define Opposite       4
#define XtSMDontChange 5
```

For definition and behavior of Above, Below, TopIf, BottomIf, and Opposite, see *Xlib - C Language X Interface*. XtSMDontChange indicates that the widget wants its current stacking order preserved.

The XtMakeGeometryRequest function performs the following:

- If the parent is not a subclass of Composite, or the parent's geometry_manager is NULL, it issues an error.

- If the widget's being_destroyed field is TRUE, it returns XtGeometryNo.

- If the widget x, y, width, height and border_width fields are all equal to the requested values, it returns XtGeometryYes.

- If the widget is unmanaged or the widget's parent is not realized, it makes the changes and returns XtGeometryYes. Otherwise, XtMakeGeometryRequest calls the parent's geometry_manager procedure with the given parameters.

- If the parent's geometry manager returns XtGeometryYes and if the widget is realized, it reconfigures the widget's window, setting its size, location, and stacking order as appropriate, by calling XConfigureWindow.

- If the geometry manager returns XtGeometryDone, it means that it has approved the change and, furthermore, has already done it. it does no configuring and changes the return value into XtGeometryYes. XtMakeGeometryRequest never returns XtGeometryDone.

- Finally, XtMakeGeometryRequest returns the resulting value from the parent's geometry manager.

10.10.2 Making Resize Requests

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

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

w Specifies the widget.

width Specifies the desired widget width.

height Specifies the desired widget height.

width_return Returns the allowed widget width.

height_return Returns the allowed widget height.

XtMakeResizeRequest is a simple interface to XtMakeGeometryRequest. It creates a XtWidgetGeometry structure and specifies that width and height should change. The geometry manager is free to modify any of the other window attributes (position or stacking order) in order to satisfy the resize request. If the return value is XtGeometryAlmost, replyWidth and replyHeight contain a "compromise" width and height. If these are acceptable the widget should immediately make an XtMakeResizeRequest requesting the compromise width and height.

If the widget is not interested in XtGeometryAlmost replies, it can pass NULL for replyWidth and replyHeight.

10.10.3 Management of Child Geometry

The geometry_manager procedure for a composite widget class is of type XtGeometryHandler:

```
typedef XtGeometryResult (*XtGeometryHandler)();

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

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

A zero (0) bit in the request's mask field means that the child widget does not care about the value of the corresponding field. Then, the geometry manager can change it as it wishes. A one (1) bit 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, it updates the widget's x, y, width, height, and border_width appropriately, and then returns `XtGeometryYes.` The value of the preferred_return argument is undefined. The widget's window is moved and resized automatically by `XtMakeGeometryRequest.`

Homogeneous composite widgets often find it convenient to treat the widget making the request the same as any other widget, possibly reconfiguring it as part of its layout process. If it does this, it should return `XtGeometryDone` to inform `XtMakeGeometryRequest` that it does not need to do the configure itself. Although `XtMakeGeometryRequest` resizes the widget's window, it does not call the widget class's resize procedure if the geometry manager returns `XtGeometryYes.` The requesting widget must perform whatever resizing calculations are needed explicitly.

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

Sometimes the geometry manager cannot satisfy the request exactly, but it may be able to satisfy what it considers 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 preferred_return with the actual changes it is willing to make, including a appropriate mask, and returns `XtGeometryAlmost.` If a bit in reply.request_mode is zero (0), the geometry manager will not change the corresponding value if the preferred_return is used immediately in a new request. If a bit is one (1), the geometry manager will change that element to the corresponding value in preferred_return. More bits may be set in preferred_return than in the original request if the geometry manager intends to change other fields should the child accept the compromise.

When `XtGeometryAlmost` is returned, the widget must decide if the compromise suggested in preferred_return is acceptable. If so, 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 preferred_return box filled in by an `XtGeometryAlmost` return and if there have been no intervening geometry requests on either its parent or any of its other children, the geometry manager must grant the request. That is, if the child asks again right away with the returned geometry, it will get an answer of `XtGeometryYes.`

To return an XtGeometryYes, the geometry manager will frequently rearrange the position of other managed children. To do this, it should call the procedure XtMoveWidget described below. However, a few geometry managers sometimes may change the size of other managed children. To do this, they should call the procedures XtResizeWidget or XtConfigureWidget.

Geometry managers must not assume that the request and preferred_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.

10.10.4 Moving and Resizing Widgets

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

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

w Specifies the widget.

x
y Specifies the new widget coordinates.

XtMoveWidget writes the new x and y values into the widget and, if the widget is realized, issues an XMoveWindow call on the widget's window.

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

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

w Specifies the widget.

width
height
border_width Specify the new widget size.

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

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

A geometry manager must not call XtResizeWidget on the child that is making the request.

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

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

w Specifies the widget.

x
y Specify the new widget position.

width
height
border_width Specify the new widget size.

XtConfigureWidget writes the new x, y, width, height, and border_width values into the widget and, if the widget is realized, makes an XConfigureWindow call on the widget's window.

If the new width or height are different from the old values, XtConfigureWidget calls the widget's resize procedure to notify it of the size change. Otherwise, it simply returns.

10.10.5 Querying Preferred Geometry

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

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

w Specifies the child widget.

intended Specifies any changes the parent plans to make to the child's geometry (can be NULL).

preferred_return Returns the child widget's preferred geometry.

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

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

The query_geometry procedure is of type `XtGeometryHandler`.

```
XtGeometryResult QueryGeometry(w, request, preferred_return)
      Widget w;
      XtWidgetGeometry *request;
      XtWidgetGeometry *preferred_return;
```

The query_geometry procedure is expected to examine the bits set in intended->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 field 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 is different from the corresponding field in intended or if a bit was set in preferred_return that was not set in intended, the query_geometry procedure should return `XtGeometryAlmost`. If the preferred geometry is identical to the current geometry, the query_geometry procedure should return `XtGeometryNo`.

After calling the query_geometry proc or if the query_geometry field is NULL, `XtQueryGeometry` 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`. `XtQueryGeometry` then returns the value returned by the query_geometry procedure or `XtGeometryYes` if the query_geometry field is NULL.

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

In addition, whether or not the caller ignores the return value or the reply mask, it is guaranteed that the reply structure contains complete geometry information for the child.

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

10.10.6 Management of Size Changes

A child can be involuntarily resized by its parent at any time. Widgets usually want to know when they have changed size so that they can re-layout their displayed data to match the new size. When a parent resizes a child, it calls XtResizeWidget. This function 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 is of type XtWidgetProc:

```
void Resize(w)
     Widget w;
```

w Specifies the widget.

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 only occur 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.

10.11 Selections

Arbitrary widgets (possibly not all in the same application) communicate with each other by means of the selection mechanism defined by the server and Xlib. For further information, see *Programming with Xlib*.

Resource Management 11

Writers of widgets need to obtain a large set of resources at widget creation time. Some of the resources come from the resource database, some from the argument list supplied in the call to XtCreateWidget, and some from the internal defaults specified for the widget. Resources are obtained first from the argument list, then from the resource database for all resources not specified in the argument list, and lastly from the internal default, if needed.

A resource is a field in the widget record with a corresponding resource entry in the widget's resource list (or in a superclass's resource list). This means that the field is settable by XtCreateWidget (by naming the field in the argument list), by an entry in the default resource files (by using either the name or class), and by XtSetValues. In addition, it is readable by XtGetValues.

Not all fields in a widget record are resources. Some are for "bookkeeping" use by the generic routines (like managed and being_destroyed). Other can be for local bookkeeping, while still others are derived from resources (many GCs and Pixmaps).

11.1 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, as well as a default value that the field should get if no value is specified. The declaration for the XtResource structure is:

```
typedef struct {
      String resource_name;
      String resource_class;
      String resource_type;
      Cardinal resource_size;
      Cardinal resource_offset;
      String default_type;
      caddr_t default_address;
} XtResource, *XtResourceList;
```

The resource_name field contains the name used by clients to access the field in the widget. By convention, it starts with a lower-case letter and is spelled almost identically to the field name, except (underbar, character) is replaced by (capital character). For example, the resource name for background_pixel is "backgroundPixel". Widget header files typically contain a symbolic name for each resource name. All resource names, classes, and types used by the Intrinsics are named in the file `<X11/StringDefs.h>`. The Intrinsic's symbolic resource names begin with `XtN` and are followed by the string name (for example, XtNbackgroundPixel for "backgroundPixel").

A resource class offers two functions:

- It isolates you from different representations that widgets can use for a similar resource.
- It lets you specify values for several actual resources with a single name. A resource class should be chosen to span a group of closely-related fields.

For example, a widget can have several pixel resources: background, foreground, border, block cursor, pointer cursor, and so on. Typically, the background defaults to "white" and everything else to "black". The resource class for each of these resources in the resource list should be chosen so that it takes the minimal number of entries in the resource database to make background "offwhite" and everything else "darkblue".

In this case, the background pixel should have a resource class of `Background` and all the other pixel entries a resource class of `Foreground`. Then, the resource file needs just two lines to change all pixels to "offwhite" or "darkblue":

```
Background:         offwhite
Foreground:         darkblue
```

Similarly, a widget may have several resource fonts (such as normal and bold), but all fonts should have the class Font. Thus, to change all fonts, simply requires a single line in the default file:

```
Font: 6x13
```

By convention, resource classes are always spelled starting with a capital letter. Their symbolic names are preceded with `XtC` (for example, XtCBackground).

The resource_type field is the physical representation type of the resource. By convention, it starts with an 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 default resource format (just about anything, but often String) to the desired physical representation (see "Resource Conversions"). The Intrinsics define the following resource types:

```
XtRBoolean
XtRLongBoolean
XtRCallback
XtRColor
XtRCursor
XtRDefaultColor
XtRDisplay
XtREditMode
XtRFile
XtRFloat
XtRFont
XtRFontStruct
XtRFunction
XtRGeometry
XtRInt
XtRJustify
XtROrientation
XtRPixel
XtRPixmap
XtRPointer
XtRString
XtRStringTable
XtRTranslationTable
XtRWidget
XtRWindow
```

The resource_size field is the size of the physical representation in bytes and normally should be specified as "sizeof(*type*)" so that the compiler fills in the value.

The resource_offset is the offset in bytes of the field within the widget. The `XtOffset` macro should be used to retrieve this value.

The default_type field is the representation type of the default resource value. If default_type is different from resource_type and the default_type is needed, the resource manager invokes a conversion procedure from default_type to resource_type. Whenever possible, the default type should be identical to the resource type in order to minimize widget creation time.

The default_address field is the address of the default resource value. The default is used only if a resource is not specified in the argument list or in the resource database.

The routines `XtSetValues` and `XtGetValues` also use the resource list to set and get widget state. For further information, see "Obtaining Widget State" and "Setting Widget State".

Here is an abbreviated version of the resource list in the Label widget:

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

}
```

The complete resource name for a field of a widget instance is the concatenation of the application name (from argv[0]) or the -name command-line option (see `XtInitialize`), the instance names of all the widget's parents, the instance name of the widget itself, and the resource name of the specified field of the widget. Likewise, the full resource class of a field of a widget instance is the concatenation of the application class (from `XtInitialize`), the widget class names of all the widget's parents (not the superclasses), the widget class name of the widget itself, and the resource name of the specified field of the widget.

11.2 Determining the Byte Offset

To determine the byte offset of a field within a structure, use `XtOffset`.

```
Cardinal XtOffset(pointer_type, field_name)
      Type pointer_type;
      Field field_name;
```

pointer_type Specifies a type that is declared as a pointer to the structure.

field_name Specifies the name of the field for which to calculate the byte offset.

`XtOffset` is usually used to determine the offset of various resource fields from the beginning of a widget.

11.3 Determining the Number of Elements

To determine the number of elements in a fixed-size array, use `XtNumber`.

```
Cardinal XtNumber(array)
      ArrayVariable array;
```

array Specifies a fixed-size array.

`XtNumber` is used to pass the number of elements in argument lists, resources lists, and other counted arrays.

11.4 Superclass to Subclass Chaining of Resource Lists

The procedure `XtCreateWidget` gets resources as a "superclass-to-subclass" operation. That is, the resources specified in Core's resource list are fetched, then those in the subclass, and so on down to the resources specified for this widget's class.

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, just 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.

11.5 Obtaining Subresources

A widget does not do anything to get its own resources. Instead, `XtCreateWidget` does this automatically before calling the class initialize procedure.

Some widgets have subparts that are not widgets but for which the widget would like to fetch resources. For example, the Text widget fetches resources for its source and sink. Such widgets call `XtGetSubresources` to accomplish this.

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

w Specifies the widget that wants resources for a subpart.

base Specifies the base address of the subpart data structure where the resources should be written.

name Specifies the name of the subpart.

class Specifies the class of the subpart.

resources Specifies the resource list for the subpart.

num_resources Specifies the number of resources in the resource list.

args Specifies the argument list to override resources obtained from the resource database.

num_args Specifies the number of arguments in the argument list. If the specified args is NULL, num_args must be zero (0). However, if num_args is zero (0), args is not referenced.

XtGetSubresources constructs a name/class list from the application name/class, the name/classes of all its ancestors, and the widget itself. Then, it appends to this list the name/class pair passed in. The resources are fetched from the argument list, the resource database, or the default values in the resource list. Then, they are copied into the subpart record.

11.6 Obtaining Application Resources

To retrieve resources that are not specific to a widget but apply to the overall application, use XtGetApplicationResources.

```
void XtGetApplicationResources(w, base, resources, num_resources, args, num_args)
      Widget w;
      caddr_t base;
      XtResourceList resources;
      Cardinal num_resources;
      ArgList args;
      Cardinal num_args;
```

w Is currently ignored and can be specified as NULL.

base Specifies the base address of the subpart data structure where the resources should be written.

resources Specifies the resource list for the subpart.

num_resources Specifies the number of resources in the resource list.

args Specifies the argument list to override resources obtained from the resource database.

num_args Specifies the number of arguments in the argument list. If the specified args is NULL, num_args must be zero (0). However, if num_args is zero (0), args is not referenced.

XtGetApplicationResources first reconstructs the application name and class and then retrieves the resources from the argument list, the resource database, or the resource list default values. After adding base to each address, the resources are copied into the address given in the resource list.

11.7 Resource Conversions

The X Toolkit provides a mechanism for registering representation converters that are automatically invoked by the resource fetching routines. The X Toolkit additionally provides and registers several commonly used converters.

This resource conversion mechanism serves several purposes:

- It permit user and application resource files to contain ASCII representations of non-textual values.
- It allows textual or other representations of default resource values that are dependent upon the display, screen, or color map, and thus must be computed at run-time.
- It caches all conversion source and result data. Conversions that require much computation or space (for example, string to translation table), or that require round trips to the server (for example, string to font or color) are performed only once.

11.7.1 Predefined Resource Converters

The X Toolkit defines all the representations used in the Core, Composite, Constraint, and Shell widgets. Furthermore, it registers resource converters from string to all these representations.

The X Toolkit registers converters for `XtRString` to the following representations:

`XtRBoolean`
`XtRLongBoolean`
`XtRCursor`
`XtRDisplay`
`XtRFile`
`XtRFont`
`XtRFontStruct`
`XtInt`
`XtPixel`

11.7.2 Writing a New Resource Converter

Type converters use pointers to `XrmValue` structures (defined in `X11/Xresource.h`) for input and output values.

```
typedef struct {
      unsigned int size;
      caddr_t addr;
} XrmValue, *XrmValuePtr;
```

A resource converter is a procedure of type XtConverter:

```
typedef void (*XtConverter)();

void Converter(args, num_args, from, to)
      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. For example, the string to font converter needs the widget's screen, or the string to pixel converter needs the widget's screen and color map. This argument is often NULL.

num_args Specifies the number of additional XrmValue arguments. This argument is often 0.

from Specifies the value to convert.

to Specifies the descriptor to use to return the converted value.

Type converters should perform the following actions:

- Check to see that the number of arguments passed is correct.
- Attempt the type conversion.
- If successful, return a pointer to the data in the to parameter. Otherwise, optionally call XtWarning and return.

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 args.

A type converter can invoke another type converter. This allows differing sources which may convert into a common intermediate result to make maximum use of the type converter cache.

Note that the address written to to.addr cannot be a local variable of the converter because this will disappear when the converter returns. It should be a pointer to a static variable, as in the following example where screenColor is returned.

The following is an example of a converter that takes a string and converts it to a Pixel:

```
static void CvtStringToPixel(args, num_args, fromVal, toVal)
      XrmValuePtr args;
      Cardinal *num_args;
      XrmValuePtr fromVal;
      XrmValuePtr toVal;
{
      static XColor screenColor;
      XColor exactColor;
      Screen *screen;
      Colormap colormap;
      Status status;
      char message[1000];

      if (*num_args != 2)
      XtError("String to pixel conversion needs screen and colormap arguments");

      screen = *((Screen **) args[0].addr);
      colormap = *((Colormap *) args[1].addr);

      status = XAllocNamedColor(DisplayOfScreen(screen), colormap,
        (String) fromVal->addr, &screenColor, &exactColor);
      if (status == 0) {
        sprintf(message, "Cannot allocate colormap entry for %s",
          (String) fromVal->addr);
      XtWarning(message);
      } else {
        (*toVal).addr = &(screenColor.pixel);
        (*toVal).size = sizeof(Pixel);
      }
};
```

11.7.3 Registering a New Resource Converter

To register a new converter, use the procedure XtAddConverter.

```
void XtAddConverter(from_type, to_type, converter, convert_args, num_args)
    String from_type;
    String to_type;
    XtConverter converter;
    XtConvertArgList convert_args;
    Cardinal num_args;
```

from_type Specifies the source type.

to_type Specifies the destination type.

converter Specifies the type converter procedure.

convert_args Specifies how to compute the additional arguments to the converter. Most type converters have none, so convert_args is NULL.

num_args Specifies the number of additional arguments to the converter. Most type converters have none, so num_args is 0.

For the few type converters that need additional arguments, the X Toolkit 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 the <X11/Convert.h> header file.

```
typedef enum {
      /* address mode          parameter representation */
         XtAddress,            /* address */
         XtBaseOffset,         /* offset */
         XtImmediate,          /* constant */
         XtResourceString,     /* resource name string */
         XtResourceQuark       /* resource name quark */
} XtAddressMode;

typedef struct {
      XtAddressMode address_mode;
      caddr_t address_id;
      Cardinal size;
} XtConvertArgRec, *XtConvertArgList;
```

The address_mode field specifies how the address_id field should be interpreted. XtAddress causes address_id to be interpreted as the address of the data. XtBaseOffset causes address_id to be interpreted as the offset from the widget base. XtImmediate causes address_id to be interpreted as a 4-byte constant. XtResourceString causes address_id to be interpreted as the name of a resource that is to be converted into an offset from widget base. XtResourceQuark is an internal compiled form of an XtResourceString.

The size field specifies the length of the data in bytes.

Here is the code used to register the CvtStringToPixel routine shown above:

```
static XtConvertArgRec colorConvertArgs[] = {
      {XtBaseOffset, (caddr_t) XtOffset(Widget, core.screen),  sizeof(Screen *)},
      {XtBaseOffset, (caddr_t) XtOffset(Widget, core.colormap),sizeof(Colormap)}
};

XtAddConverter(XtRString, XtRPixel, CvtStringToPixel,
    colorConvertArgs, XtNumber(colorConvertArgs));
```

The conversion argument descriptors colorConvertArgs and screenConvertArg are predefined for you. 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].

It might seem easier to just create a descriptor that puts the widget's base address into args[0], and do your own indexing off that in the conversion routine. But you should not. If you constrain the dependencies of your conversion procedure to the minimum possible, you improve the chance that subsequent conversions will find what they need in the conversion cache. Then, you decrease the size of the cache by having fewer but more widely applicable entries.

11.7.4 Invoking Resource Converters

All resource-fetching routines (for example, `XtGetSubresources`, `XtGetApplicationResources`, and so on) call resource converters if the user specifies a resource that is a different representation from the desired representation, or if the widget's default resource value representation is different from the desired representation.

To invoke resource conversions, use `XtConvert` or `XtDirectConvert`.

The definition for `XtConvert` is:

```
void XtConvert(w, from_type, from, to_type, to_return)
    Widget w;
    String from_type;
    XrmValuePtr from;
    String to_type;
    XrmValuePtr to_return;
```

w	Specifies the widget to use for additional arguments (if any are needed).
from_type	Specifies the name of the source type.
from	Specifies the value to be converted.
to_type	Specifies the name of the destination type.
to_return	Returns the converted value.

`XtConvert` looks up the type converter registered to convert from_type to to_type and computes any additional arguments needed. It then calls `XtDirectConvert`.

The definition for `XtDirectConvert` is:

```
void XtDirectConvert(converter, args, num_args, from, to_return)
    XtConverter converter;
    XrmValuePtr args;
    Cardinal num_args;
    XrmValuePtr from;
    XrmValuePtr to_return;
```

converter Specifies the widget to use for additional arguments (if any are needed).

args Specifies the additional arguments needed to perform the conversion (often NULL).

num_args Specifies the number of additional arguments (often 0).

from Specifies the value to be converted.

to_return Returns the converted value.

`XtDirectConvert` looks in the converter cache to see if this conversion procedure has been called with the specified arguments. If so, it just returns a descriptor for information stored in the cache. Otherwise, it calls the converter and enters the result in the cache.

11.8 Reading and Writing Widget State

Any resource field in a widget can be read or written by a client. On a write, the widget decides what changes it will actually allow and updates all derived fields appropriately.

11.8.1 Obtaining Widget State

To retrieve the current value of a resource associated with a widget instance, use `XtGetValues`.

```
void XtGetValues(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

w Specifies the widget.

args Specifies a variable length argument list of name/address pairs that contain the resource name and the address to store the resource value into. The argument names in args are dependent on the widget.

num_args Specifies the number of arguments in argument list.

`XtGetValues` starts with the resources specified for the core widget fields and proceeds down the subclass chain to the widget.

The value field of a passed Arg should contain the address into which to store the corresponding resource value.

If the widget's parent is a subclass of `constraintWidgetClass`, `XtGetValues` then fetches the values for any constraint resources requested. It starts with the constraint resources specified for `constraintWidgetClass` and proceeds down to the subclass chain to the parent's constraint resources.

Finally, the get_values_hook procedures, if non-NULL, are called in superclass-to-subclass order after all the resource values have been fetched by `XtGetValues`. This permits a subclass to provide non-widget resource data by means of the GetValues mechanism.

Widget Subpart Resource Data

Widgets that have subparts can return the resource values by using `XtGetValues` and supplying a get_values_hook procedure. The get_values_hook procedure is of type `XtArgsProc`:

```
void get_values_hook(w, args, num_args)
     Widget w;
     ArgList args;
     Cardinal *num_args;
```

w Specifies the widget whose non-widget resource values are to be retrieved.

args Specifies the argument list that was passed to `XtCreateWidget`.

num_args Specifies the number of arguments in the argument list.

Obtaining Widget Subpart State

To retrieve the current value of a non-widget resource data associated with a widget instance, use `XtGetSubvalues`. For a discussion of non-widget subclass resources resources, see "Obtaining Subresources".

```
void XtGetSubvalues(base, resources, num_resources, args, num_args)
      caddr_t base;
      XtResourceList resources;
      Cardinal num_resources;
      ArgList args;
      Cardinal num_args;
```

base Specifies the base address of the subpart data structure from which the resources should be retrieved.

resources Specifies the non-widget resources list.

num_resources Specifies the number of resources in the resource list.

args Specifies a variable length argument list of name/address pairs that contain the resource name and the address to store the resource value into. The arguments and values passed in args are dependent on the subpart. The storage for argument values that are pointed to by args must be deallocated by the application when no longer needed.

num_args Specifies the number of arguments in argument list.

11.8.2 Setting Widget State

To modify the current value of a resource associated with a widget instance, use `XtSetValues`.

```
void XtSetValues(w, args, num_args)
      Widget w;
      ArgList args;
      Cardinal num_args;
```

w Specifies the widget.

args Specifies a variable length argument list of name/value pairs that contain the resources to be modified and their new values. The resources and values passed are dependent on the widget being modified.

num_args Specifies the number of resources in the argument list.

`XtSetValues` starts with the resources specified for the core widget fields and proceeds down the subclass chain to the widget. At each stage, it writes the new value (if specified by one of the arguments) or the existing value (if no new value is specified) to a new widget data record.

`XtSetValues` then calls the set_values procedures for the widget in "superclass-to-subclass" order. If the widget has non-NULL set_values_hook fields, these are called with the arguments immediately after the corresponding set_values procedure. This procedure permits subclasses to set non-widget data using the SetValues mechanism.

If the widget's parent is a subclass of `constraintWidgetClass`, `XtSetValues` also updates the widget's constraints. It starts with the constraint resources specified for `constraintWidgetClass` and proceeds down the subclass chain to the parent's class. At each stage, it writes the new value or the existing value to a new constraint record. It then calls the constraint set_values procedures from `constraintWidgetClass` down to the parent's class. The constraint set_values procedures are called with widget arguments (as for all set_values procs, not just the constraint record arguments), so that they can make adjustments to the desired values based on full information about the widget.

XtSetValues determines if a geometry request is needed by comparing the current widget to the new widget. If any geometry changes are required, it makes the request, and the geometry manager returns XtGeometryYes, XtGeometryAlmost, or XtGeometryNo. If XtGeometryYes, XtSetValues calls the widget's resize procedure. If XtGeometryNo, XtSetValues resets the geometry fields to their original values. If XtGeometryAlmost, XtSetValues calls the set_values_almost procedure, which determines what should be done and writes new values for the geometry fields into the new widget. XtSetValues then repeats this process, deciding once more whether the geometry manager should be called.

Finally, if any of the set_values procedures returned TRUE, XtSetValues causes the widget's expose procedure to be invoked by calling XClearArea on the widget's window.

Widget State

The set_values procedure for a widget class is of type XtSetValuesFunc:

```
typedef Boolean (*XtSetValuesFunc)();

Boolean SetValuesFunc(current, request, new)
      Widget current;
      Widget request;
      Widget new;
```

current Specifies the existing widget.

request Specifies a copy of the widget asked for by the XtSetValues call before any class set_values procedures have been called.

new Specifies a copy of the widget with the new values that are actually allowed.

The set_values procedure should recompute any field derived from resources that are changed (for example, many GCs depend upon foreground and background). If no recomputation is necessary and if none of the resources specific to a subclass require the window to be redisplayed when their values are changed, then 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. In this case, though, the "reference" widget is "request", not "new".

"New" starts with the values of "request" but has been modified by any superclass set_values procedures. A widget need not refer to "request" unless it must resolve conflicts between "current" and "new". Any changes that the widget wishes to make should be made in "new". XtSetValues will copy the "new" values back into the "current" widget instance record after all class set_values procedures have been called.

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 will force 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.

It is permissible to call XtSetValues before a widget is realized. Therefore, the set_values proc must not assume that the widget is realized.

Widget State

The set_values_almost procedure for a widget class is of type XtAlmostProc:

```
typedef void (*XtAlmostProc)();

void AlmostProc(w, new_widget, request, reply)
      Widget w;
      Widget new_widget;
      XtWidgetGeometry *request;
      XtWidgetGeometry *reply;
```

w	Specifies the widget on which the geometry change is requested.
new_widget	The return value, with relevant geometry fields modified based on the geometry requests.
request	Specifies the original geometry request that was sent to the geometry manager that returned XtGeometryAlmost.
reply	Specifies the compromise geometry that was returned by the geometry manager that returned XtGeometryAlmost.

Most classes inherit this operation from their superclass by copying the Core set_values_almost procedure in their class_initialize procedure. The Core's set_values_almost procedure simply accepts the compromise suggested.

The set_values_almost procedure is called when a client tries to set a widget's geometry by means of a call to XtSetValues, and the geometry manager cannot satisfy the request but instead returns XtGeometryAlmost and a compromise geometry. The set_values_almost procedure takes the original geometry and the compromise geometry and determines whether the compromise is acceptable or a different compromise might work. It returns its results in the new_widget parameter, which will then be sent back to the geometry manager for another try.

Widget State

The constraint set_values procedure is of type XtSetValuesFunc. The values passed to the parent's constraint set_values procedure are the same as those passed to the child's class set_values procedure. A class can specify NULL for the set_values field of the ConstraintPart if it need not compute anything.

The constraint set_values procedure should recompute any constraint fields derived from constraint resource that are changed. Further, it should modify the widget fields as appropriate. For example, if a constraint for the maximum height of a widget is changed to a value smaller than the widget's current height, the constraint set_values procedure should reset the height field in the widget.

Setting Widget Subpart State

To set the current value of a non-widget resource associated with a widget instance, use XtSetSubvalues. For a discussion of non-widget subclass resources, see "Obtaining Subresources".

```
void XtSetSubvalues(base, resources, num_resources, args, num_args)
      caddr_t base;
      XtResourceList resources;
      Cardinal num_resources;
      ArgList args;
      Cardinal num_args;
```

base Specifies the base address of the subpart data structure where the resources should be written.

resources Specifies the current non-widget resources values.

num_resources Specifies the number of resources in the resource list.

args Specifies a variable length argument list of name/value pairs that contain the resources to be modified and their new values. The resources and values passed are dependent on the subpart of the widget being modified.

num_args Specifies the number of resources in argument list.

Widget Subpart Resource Data

Widgets that have a subpart can set the resource values by using XtSetValues and supplying a set_values_hook procedure. The set_values_hook procedure for a widget class is of type XtArgsFunc:

```
typedef Boolean (*XtArgsFunc)();

Boolean ArgsFunc(w, args, num_args)
     Widget w;
     ArgList args;
     Cardinal *num_args;
```

w Specifies the widget whose non-widget resource values are to be changed.

args Specifies the argument list that was passed to `XtCreateWidget`.

num_args Specifies the number of arguments in the argument list.

Translation Management - Handling User Input 12

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 user-overridable default mapping of events into behavior.

The translation manager provides an interface to specify and manage the mapping of X Event sequences into widget-supplied functionality. The simplest example would be to call procedure Abc when key "y" is pressed.

The translation manager uses two kinds of tables to perform translations. The "action table", which is in the widget class structure, specifies the mapping of externally available procedure name strings to the corresponding procedure implemented by the widget class. The "translation table", which is in the widget class structure, specifies the mapping of event sequence to procedure names.

The translation table in the class structure can be over-ridden for a specific widget instance by supplying a different translation table for the widget instance.

12.1 Action Tables

All widget class records contain an action table. In addition, an application can register its own action tables with the translation manager, so that the translation tables it provides to widget instances can access application functionality. The translation action_proc procedure is of type `XtActionProc`:

```
typedef void (*XtActionProc)();

void ActionProc(w, event, params, num_params)
      Widget w;
      XEvent *event;
      String *params;
      Cardinal *num_params;
```

w Specifies the widget that caused the action to be called.

event Specifies the event that caused the action to be called. If the action is called after a sequence of events, then the last event in the sequence is used.

params Specifies a pointer to the list of strings that were specified in the translation table as arguments to the action.

num_params Specifies the number of arguments specified in the translation table.

```
typedef struct _XtActionsRec {
      String action_name;
      XtActionProc action_proc;
} XtActionsRec, *XtActionList;
```

The action_name field is the name that you use in translation tables to access the procedure. The action_proc field is a pointer to a procedure that implements the functionality.

For example, the Command widget has procedures to:

- Set the command button to indicate it is activated
- Unset the button back to its normal mode
- Highlight the button borders
- Unhighlight the button borders
- Notify any callbacks that the button has been activated

The action table for the Command widget class makes these functions available to translation tables written for Command or any subclass. The string entry is the name used in translation tables. The procedure entry (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},
};
```

12.1.1 Registering Action Tables

To make functionality available by declaring an action table and registering this with the translation manager, use `XtAddActions`.

```
void XtAddActions(actions, num_actions)
      XtActionList actions;
      Cardinal num_actions;
```

actions Specifies the action table to register.

num_args Specifies the number of entries in actions.

The X Toolkit registers an action table for MenuPopup and MenuPopdown as part of X Toolkit initialization.

12.1.2 Translating Action Names to Procedures

The translation manager uses a simple algorithm to convert the name of procedure specified in a translation table into the actual procedure specified in an action table. It performs a search for the name in the following tables:

- The widget's class action table for the name
- The widget's superclass action table, and on up the superclass chain
- The action tables registered with XtAddActions, from the most recently added table to the oldest table.

As soon as it finds a name, it stops the search. If it cannot find a name, the translation manager generates an error.

12.2 Translation Tables

All widget instance records contain a translation table, which is a resource with no default value. A translation table specifies what action procedures are invoked for an event or a sequence of events. It is a string containing a list of translations from an event (or event sequence) into one or more procedure calls. The translations are separated from one another by new-line characters (ASCII LF).

For example, the default behavior of Command is:

- Highlight on enter window
- Unhighlight on exit window
- Invert on button 1 down
- Call callbacks and reinvert on button 1 up

Command's default translation table is:

```
static String defaultTranslations =
      "<EnterWindow>:Highlight()\n\
      <LeaveWindow>:Unhighlight()\n\
      <Btn1Down>: Set()\n\
      <Btn1Up>:   Notify() Unset()";
```

For details on the syntax of translation tables, see Appendix B.

The tm_table field of the `CoreClass` record should be filled in at static initialization time with the string containing the class's default translations. If a class wishes to just inherit its superclass's translations, it can store the special value `XtInheritTranslations` into tm_table. After the class initialization procedures have been called, the Intrinsics compile this translation table into an efficient internal form. Then, at widget creation time, this default translation table will be used for any widgets that have not had their core translations field set by the resource manager or the initialize procedures.

The resource conversion mechanism takes care of automatically compiling string translation tables that are resources. If a client uses translation tables that are not resources, it must compile them itself using `XtParseTranslations`.

The X Toolkit uses 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.

12.3 Merging Translation Tables

Sometimes an application needs to destructively or non-destructively add its own translations to a widget's translation. For example, a window manager provides functions to move a window. It normally may moves the window when any pointer button is pressed down in a title bar. It allows the user to specify other translations for the middle or right button down in the title bar, but it ignores any user translations for button 1 down.

To accomplish this, the window manager first should create the title bar and then should merge the two translation tables into the title bar's translations. One translation table contains the translations that the window manager wants only if the user has not specified a translation for a particular event (or event sequence). The other translation table contains the translations that the window manager wants regardless of what the user has specified.

Three X Toolkit functions support this merging:

XtParseTranslationTable	Compiles a translation table.
XtAugmentTranslations	Merges (non-destructively) a compiled translation table into a widget's compiled translation table.
XtOverrideTranslations	Merges destructively a compiled translation table into a widget's compiled translation table.

To compile a translation table, use XtParseTranslationTable.

```
XtTranslations XtParseTranslationTable(table)
      String table;
```

table Specifies the translation table to compile.

XtParseTranslationTable compiles the translation table into the opaque internal representation (of type XtTranslations).

To merge new translations into an existing translation table, use XtAugmentTranslations.

```
void XtAugmentTranslations(w, translations)
      Widget w;
      XtTranslations translations;
```

w Specifies the widget to merge the new translations into.

translations Specifies the compiled translation table to merge in.

XtAugmentTranslations non-destructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is ignored.

To overwrite existing translations with new translations, use XtOverrideTranslations.

```
void XtOverrideTranslations(w, translations)
      Widget w;
      XtTranslations translations;
```

w	Specifies the widget to merge the new translations into.
translations	Specifies the compiled translation table to merge in.

`XtOverrideTranslations` destructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is merged in and override the widget's translation.

Resource File Format A

A resource file contains text representing the default resource values for an application or set of applications. The resource file is an ASCII text file that consists of a number of lines with the following EBNF syntax:

```
Xdefault       = {line "\\n"}.
line           = (comment | production).
comment        = "!" string.
production     = resourcename ":" string.
resourcename   = ["*"] name {("." | "*") name}.
string         = {<any character not including eol>}.
name           = {"A"-"Z" | "a"-"z" | "0"-"9"}.
```

If the last character on a line is a backslash (\), that line is assumed to continue on the next line.

To include a new-line character in a string, use “\n”.

This page left blank intentionally.

Translation Table File Syntax B

A translation table file is an ASCII text file.

Notation

Syntax is specified in EBNF notation, where:

```
[ a ]
```

means either nothing or “a”.

```
{ a }
```

means 0 or more occurrences of “a”

All terminals are enclosed in "double" quotes. Informal descriptions are enclosed in <angle> brackets.

Syntax

The syntax of the translation table file is:

```
translationTable   = [ production { "\\n" production } ]
production         = lhs ":" rhs
lhs                = ( event | keyseq ) { "," (event | keyseq) }
keyseq             = """" keychar {keychar} """"
keychar            = [ "^" | "$" ] <ascii character>
event              = [modifier_list] "<"event_type">" [ "(" count["+"] ")" ] {detail}
modifier_list      = [!] modifier {modifier} | "None"
modifier           = [˜ ] modifier_name
count              = (2 | 3 | 4 | ...)
modifier_name      = <see ModifierNames table below>
event_type         = <see Event Types table below>
detail             = <event specific details>
rhs                = { name "(" [params] ")" }
name               = namechar { namechar }
namechar           = { "a"-"z" | "A"-"Z" | "0"-"9" | "$" | "_" }
params             = string {"," string}.
string             = quoted_string | unquoted_string
quoted_string      = """" {<ascii character>} """"
unquoted_string    = {<ascii character except space, tab, ",", newline, ")">}
```

Informally, the productions are an event specifier on the left (terminated with a colon) and a list of action specifications on the right (terminated with a newline).

The information on the left specifies the X Event, complete with modifiers and detail fields, while that on the right specifies what to do when that event is detected. An action is the name of an exported function. The parameters are strings.

It is often convenient to include newlines in a translation table to make it more readable. In C, the newline should be preceded by a backslash (\):

```
"<Btn1Down>: DoSomething()\n\
<Btn2Down>: DoSomethingElse()"
```

Modifier Names

The Modifier field is used to specify normal X keyboard and button modifier mask bits. If the modifier_list has no entries and is not "None", it means "don't care" on all modifiers. If any modifiers are specified, and "!" is not specified, it means that the listed modifiers must be in the correct state and "don't care" about any other modifiers. If "!" 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 a modifier is preceded by a "~ " it means that that modifier must not be asserted. If "None" is specified, it means no modifiers can be asserted. 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
```

Modifier	Meaning
c	Control Key
Ctrl	Control Key
s	Shift Key
Shift	Shift Key
m	Modifier 1
Meta	Modifier 1
l	Lock Key
Lock	Lock Key
1	Modifier 1
Mod1	Modifier 1
2	Modifier 2
Mod2	Modifier 2
3	Modifier 3
Mod3	Modifier 3
4	Modifier 4
Mod4	Modifier 4
5	Modifier 5
Mod5	Modifier 5
ANY	Any combination

Event Types

The EventType field describes XEvent types. The following are the currently defined EventType values:

Type	Meaning
Key	KeyPress
KeyDown	KeyPress
KeyUp	KeyRelease
BtnDown	ButtonPress
BtnUp	ButtonRelease
Motion	MotionNotify
BtnMotion	MotionNotify with any button down
Btn1Motion	MotionNotify with button 1 down
Btn2Motion	MotionNotify with button 2 down
Btn3Motion	MotionNotify with button 3 down
Btn4Motion	MotionNotify with button 4 down
Btn5Motion	MotionNotify with button 5 down
Enter	EnterNotify
Leave	LeaveNotify
FocusIn	FocusIn
FocusOut	FocusOut
Keymap	KeymapNotify
Expose	Expose
GrExp	GraphicsExpose
NoExp	NoExpose
Visible	VisibilityNotify
Create	CreateNotify
Destroy	DestroyNotify
Unmap	UnmapNotify
Map	MapNotify
MapReq	MapRequest
Reparent	ReparentNotify
Configure	ConfigureNotify

ConfReq	ConfigureRequest
Grav	GravityNotify
ResReq	ResizeRequest
Circ	CirculateNotify
CircReq	CirculateRequest
Prop	PropertyNotify
SelClr	SelectionClear
SelReq	SelectionRequest
Select	SelectionNotify
Clrmap	ColormapNotify
Message	ClientMessage
Mapping	MappingNotify

Supported Abbreviations:

Abbreviation	Meaning
Ctrl	KeyPress with control modifier
Meta	KeyPress with meta modifier
Shift	KeyPress with shift modifier
Btn1Down	ButtonPress with Btn1 detail
Btn1Up	ButtonRelease with Btn1 detail
Btn2Down	ButtonPress with Btn2 detail
Btn2Up	ButtonRelease with Btn2 detail
Btn3Down	ButtonPress with Btn3 detail
Btn3Up	ButtonRelease with Btn3 detail
Btn4Down	ButtonPress with Btn4 detail
Btn4Up	ButtonRelease with Btn4 detail
Btn5Down	ButtonPress with Btn5 detail
Btn5Up	ButtonRelease with Btn5 detail

The Detail field is event specific and normally corresponds to the detail field of an X Event, for example, <Key>A. If no detail field is specified, then ANY is assumed.

Useful Examples

- Always put more specific events in the table before more general ones:

  ```
  Shift <Btn1Down> : twas()\n\
  <Btn1Down> : brillig()
  ```

- For double-click on Button 1 Up with Shift, use:

  ```
  Shift<Btn1Up>(2) : and()
  ```

  This is equivalent to

  ```
  Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down>,Shift<Btn1Up> : and()
  ```

  with appropriate timers set between events.

- For double-click on Button 1 Down with Shift, use:

  ```
  Shift<Btn1Down>(2) : the()
  ```

  This is equivalent to

  ```
  Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down> : the()
  ```

  with appropriate timers set between events.

- 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 jiggles a bit between the down and up events. Similarly, any motion event specified in a translation matches any number of motion events. If the motion event causes an action procedure to be invoked, the procedure is invoked after each motion event.

- If an event sequence consists of a sequence of events that is also a non-initial subsequence of another translation, it is not taken if it occurs in the context of the longer sequence. This occurs mostly in sequences like:

  ```
  <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.

- For single click on Button 1 Up with Shift and Meta, use:

  ```
  Shift Meta <Btn1Down>, Shift Meta<Btn1Up>: gyre()
  ```

- The "+" notation allows you to say "for any number of clicks greater than or equal to count", such as:

  ```
  Shift <Btn1Up>(2+) : and()
  ```

- To say EnterNotify with any modifiers, use:

  ```
  <Enter> : gimble()
  ```

- To say EnterNotify with no modifiers, use:

  ```
  None <Enter> : in()
  ```

- To say EnterNotify with Button 1 Down and Button 2 Up and don't care about the other modifiers, use:

  ```
  Button1 ~Button2 <Enter> : the()
  ```

- To say EnterNotify with Button1 Down and Button2 Down exclusively, use:

  ```
  ! Button1 Button2 <Enter> : wabe()
  ```

  It is never necessary to use ~ with !.

This page left blank intentionally.

Conversion Notes C

1. In the alpha release 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 this X Toolkit, a single generic `XtCreateWidget` performs most of the common work and then calls the initialize procedure implemented for the particular widget class.

2. Each composite widget class also implemented the procedures Xt<*Widget*>Add and an Xt<*Widget*>Delete (for example, `XtButtonBoxAddButton` and `XtButtonBoxDeleteButton`). In the beta release X Toolkit, the composite generic procedures `XtManageChildren` and `XtUnmanageChildren` perform error-checking and screening out of certain children. Then, they call the change_managed procedure implemented for the widget's composite class. If the widget's parent has not yet been realized, the call on the change_managed procedure is delayed until realization time.

3. The new X Toolkit can be used to implement old-style calls by defining one-line procedures or macros that invoke a generic routine. For example, you could define the macro `XtCreateLabel`: as the :

```
#define XtCreateLabel(name, parent, args, num_args) \
        ((LabelWidget) XtCreateWidget(name, labelWidgetClass,    parent, args, num_args))
```

Index