X Toolkit Library - C Language Interface

Toolkit Beta Version 0.1

X Protocol Version 11

The X Window System is a trademark of MIT.

1. Overview 1. Overview

The X Toolkit provides the base functionality necessary to build a wide variety of application environments. The X Toolkit is fully extensible and supportive of the independent development of new or extended components. This is accomplished by defining a few, easy-to-use 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. As time progresses, we expect to supply a reasonably complete and cohesive set of sample widgets that can be used as the basis for quick applications development. In the X Toolkit, a ‘‘widget’’ is a combination of an X window (or subwindow) and its associated semantics.

The X Toolkit provides flexibility and extensibility. To the extent possible, it also 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.

2. Introduction 2. 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 that enable you to manage:

- Widgets
- Standard operations
- Toolkit initialization
- Memory
- Input and events
- Widget geometry
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmaps
- Errors

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 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 toolkit functions are:

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

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

- <X/Xlib.h>
- <X/Intrinsic.h>
- <X/Xatoms.h>
- <X/Atoms.h>
- <X/TopLevel.h>

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

3. Terminology

The following terms are used throughout this manual:

Object	A software data abstraction that consists of private data and public and private 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.
Widget	An object providing a user-interface abstraction (for example, a scroll bar).
Toolkit	The name of the user-interface library that is the subject of this manual.
Application Programmer	A programmer that uses the X Toolkit to produce the application user-interface.
Widget Programmer	A programmer that modifies the X Toolkit or adds new widgets to it.
Client	A routine that uses a widget in an application or for composing another widget.
Name	The name that is specific to a given widget instance in an application.
Class	A general group that a widget belongs to.

4. Widgets 4. Widgets

The fundamental data type of the 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.

Physically, a widget is a pointer to a structure which contains fields common to all widgets and some fields that are specific to widgets of a particular widget class. In general, the values of these fields may vary from widget to widget, even for widgets of the same widget class. A widget is allocated and initialized by XtCreateWidget. See ‘‘Creating Widgets’’ for further information.

Physically, a widget class is a pointer to a structure which contains fields common to all widget classes and some fields that are particular to the widget class. The values of these fields are constant for all widgets of the widget class, though the values may vary from widget class to widget class. (‘‘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.)

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 Subclassing’’. The predefined widget classes below adhere to these conventions.

4.1. Core Widget Definitions 4.1. Core Widget Definitions

The Core widget contains the generic widget definitions. All actual widgets are subclasses of Core. The structure definitions for Core, Composite and Constraint currently are contained in the Intrinsic.h file.

4.1.1. CorePart Definition 4.1.1. CorePart Definition

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

```
typedef struct {
        WidgetClass widget_class;           See ‘‘Widget Classes’’
        Widget parent;                      See ‘‘Widget Tree’’
        String name;                        See ‘‘Resource Management’’
        XrmName xrm_name;                   Private to ‘‘Resource Management’’
        Screen *screen;                     See ‘‘Window Attributes’’
        Window window;                      See ‘‘Window Attributes’’
        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 ‘‘Window Attributes’’
        Pixmap border_pixmap;               See ‘‘Window Attributes’’
        Pixel background_pixel;             See ‘‘Window Attributes’’
        Pixmap background_pixmap;           See ‘‘Window Attributes’’
        _XtEventTable event_table;          Private to ‘‘Event Management’’
        _XtTranslations translations;       Private to ‘‘Translation Management’’
        Boolean visible;                    See ‘‘Widget Visibility and Exposure’’
        Boolean sensitive;                  See ‘‘XtSetSensitive’’
        Boolean ancestor_sensitive;         See ‘‘XtSetSensitive’’
        Boolean managed;                    See ‘‘Composite Widgets’’
        Boolean mapped_when_managed;        See ‘‘Composite Widgets’’
        Boolean being_destroyed;            See ‘‘Destroying Widgets’’
        XtCallbackList destroy_callbacks;   See ‘‘Destroying Widgets’’
} CorePart;
```

All widgets have the core fields as their first component. The prototypical type Widget is defined with only this set of fields: various routines may cast these to specific widget types as needed.

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

4.1.2. CorePart Default Values 4.1.2. CorePart Default Values

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

Field	Default Value
widget_class	widget_class argument to XtCreateWidget
parent	parent argument to XtCreateWidget
name	name argument to XtCreateWidget
screen	parent's screen field (top-level widget from display specifier)
window	NULL
x	0
y	0
width	0
height	0
depth	parent's depth field (top-level widget from root window)
border_width	1
border_pixel	black
border_pixmap	NULL
background_pixel	white
background_pixmap	NULL
visible	TRUE
sensitive	TRUE
ancestor_sensitive	parent's (sensitive & ancestor_sensitive)
managed	FALSE
map_when_managed	TRUE
being_destroyed	parent's being_destroyed
destroy_callbacks	NULL

4.1.3. CoreClassPart Definitions 4.1.3. CoreClassPart Definitions

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

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

typedef struct {
          WidgetClass superclass;              See ‘‘Widget Subclassing’’
          String class_name;                   See ‘‘Widget Subclassing’’
          Cardinal widget_size;                See ‘‘Creating Widgets’’
          XtProc class_initialize;             See ‘‘Widget Subclassing’’
          Boolean class_inited;                Private to ‘‘XtCreateWidget’’
          XtInitProc initialize;               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 visible_interest;            See ‘‘Widget Visibility and Exposure’’
          XtWidgetProc destroy;                See ‘‘Destroying Widgets’’
          XtWidgetProc resize;                 See ‘‘Geometry Management’’
          XtExposeProc expose;                 See ‘‘Widget Visibility and Exposure’’
          XtSetValuesProc set_values;          See ‘‘Reading and Writing Widget State’’
          XtWidgetProc accept_focus;           See ‘‘Focus 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. That is, various routines may cast these to specific widget class types as needed:

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

The predefined class record and pointer for WidgetClassRec are:

```
extern WidgetClassRec widgetClassRec;

extern WidgetClass widgetClass;
```

Applications that do not define new widget classes usually do not need to access the internals of widgets. The opaque types Widget and WidgetClass and the opaque variable widgetClass are defined for generic actions on widgets. Applications can use these types for generality or can use the widget-specific types to make the code indicate external representation.

4.2. Composite Widget Definitions 4.2. Composite Widget Definitions

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

4.2.1. CompositePart Definition 4.2.1. CompositePart Definition

Composite widgets have the following additional fields defined in the CompositePart structure:

```
typedef struct {
        WidgetList children;                See ``Widget Tree''
        Cardinal num_children;              See ``Widget Tree''
        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;
```

4.2.2. CompositePart Default Values 4.2.2. CompositePart Default Values

The default values for the composite fields (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

4.2.3. CompositeClassPart Definitions 4.2.3. CompositeClassPart Definitions

Composite widget classes have the following additional fields:

```
typedef struct {
        XtGeometryHandler geometry_manager;See ``Geometry Management''
        XtWidgetProc change_managed;        See ``Composite Widgets''
        XtArgsProc 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 CompositeWidgetClass *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.

To determine if a widget *w* is composite (or a subclass), use the Boolean function

XtIsComposite.

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

4.3. Constraint Widget Definitions 4.3. Constraint Widget Definitions

Constraint widgets are a subclass of the Composite widget, and are more fully described in ‘‘Constrained Composite Widgets’’.

4.3.1. ConstraintPart Definition 4.3.1. ConstraintPart Definition

Constraint widgets have the following additional fields defined in the ConstraintPart structure:

```
typedef struct {
        caddr_t mumble;                 No new fields, keep C compiler happy
} ConstraintPart;
```

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

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

4.3.2. ConstraintClassPart Definitions 4.3.2. ConstraintClassPart Definitions

Constraint widget classes have the following additional fields:

```
typedef struct {
        XtResourceList constraints;     See ‘‘Constrained Composite Widgets’’
        Cardinal num_constraints;       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 ConstraintWidgetClass constraintWidgetClass;
```

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

5. Widget Classes 5. Widget Classes

The widget_class field of a widget points to its widget class structure. The widget class 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. That is, they implement procedures that are available through their widget class structure. These class procedures are invoked by generic procedures, which envelope 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.

Note

> In the alpha release 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 toolkit, a single generic XtCreateWidget performs most of the common work and then calls the initialize procedure implemented for the particular widget class.
>
> Each composite widget class also implemented the procedures Xt<*Widget*>Add and an Xt<*Widget*>Delete (for example, XtButtonBoxAddButton and XtButtonBoxDeleteButton). In the beta release toolkit, the composite generic procedures XtManageChildren and XtUnmanageChildren perform error-checking and screening out of certain children. Then, they call the changed_managed procedure implemented for the widget's composite class.
>
> Note that the new toolkit can be used to implement old-style calls by defining one-line procedures or macros that invoke a generic routine. For example, you can define the macro XtCreateLabel:

```
#define XtCreateLabel(name, parent, args, num_args) \
        ((LabelWidget) XtCreateWidget(name, labelWidgetClass,    \
        parent, args, num_args))
```

6. Widget Subclassing 6. Widget Subclassing

All widget classes are a subclass of the Core class and may be subclassed further. For example, Label is a direct subclass of Core. That is, it adds a text field. Command is a direct subclass of Label. In addition to a text field, it has a callback procedure that can be invoked. Toggle is a direct subclass of Command. In addition to the text and callback, it has a Boolean state that can be toggled.

Subclassing reduces the amount of code and declarations you write to make a new widget class. For example, you do not have to include every resource your widget uses in an XtResourceList, but you do have to include the resources your widget has that its superclass does not.

In order to make good use of subclassing, widget declarations are highly stylized. A widget consists of three parts:

- 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

6.1. Widget Subclassing in Public ‘‘.h’’ Files 6.1. Widget Subclassing in Public ‘‘.h’’ Files

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

- The atom names (strings) of the new resources that this widget adds to its superclass
- The class record pointer for the widget
- The C type you use to declare widgets of this class

For example, here is the public ‘‘.h’’ file for the Label widget:

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

/* Class record pointer */
extern WidgetClass labelWidgetClass;

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

6.2. Widget Subclassing in Private ‘‘.h’’ Files 6.2. 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:

- 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 widget class structure.
- The complete widget class structure for this widget.
- A ‘‘constant’’ of the generic widget class structure.

For example, here is the private ‘‘.h’’ file for the Label widget:

```
/* New fields for the Label widget record */
typedef struct {
/* Settable resources */
        Pixel foreground;
        XFontStruct *font;
        String label;
        XtJustify justify;
        Dimension internal_width;
        Dimension internal_height;

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

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

/* New fields for the Label widget class record */
typedef struct {
        int mumble;                    /* No new fields, but keep C compiler happy */
} LabelClassPart;

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

/* Class record variable */
extern LabelClassRec labelClassRec;
```

6.3. Widget Subclassing in ‘‘.c’’ Files **6.3. Widget Subclassing in ‘‘.c’’ Files**

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

- Class information (superclass, class_name, widget_size, class_initialize, class_inited)
- Data Constants (resources and num_resources, actions and num_actions, visible_interest, compress_motion, compress_expose)
- Widget Operations (initialize, realize, destroy, resize, expose, set_values, accept_focus)

The superclass field points to the superclass record. For all direct subclass of the generic core widget this means that superclass is initialized to the address of the widgetClassRec structure.

The class_name field contains the text name for this class (that is used by the resource manager). For example, the Label widget has the string ‘‘Label’’. The widget_size field is the size of the corresponding Widget structure, not the size of the Class structure. All other fields are described in their respective sections.

Here is a somewhat compressed version of ‘‘.h’’ file for the Label widget:

```
/* Resources specific to Label */
#define XtRjustify                    "Justify"
static XtResource resources[] = {
        {XtNforeground, XtCForeground, XrmRPixel, sizeof(Pixel),
          XtOffset(LabelWidget, label.foreground), XrmRString, "Black"},
        {XtNfont,  XtCFont, XrmRFontStruct, sizeof(XFontStruct *),
          XtOffset(LabelWidget, label.font),XrmRString, "Fixed"},
        {XtNlabel,  XtCLabel, XrmRString, sizeof(String),
          XtOffset(LabelWidget, label.label), XrmRString, NULL},
                                      .
                                      .
                                      .
}

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

```
/* Class record constant */
LabelClassRec labelClassRec = {
 {
/* core_class fields */
	/* superclass        */		(WidgetClass) &widgetClassRec,
	/* class_name        */		"Label",
	/* widget_size       */		sizeof(LabelRec),
	/* class_initialize  */		ClassInitialize,
	/* class_inited      */		FALSE,
	/* initialize        */		Initialize,
	/* realize           */		Realize,
	/* actions           */		NULL,
	/* num_actions       */		0,
	/* resources         */		resources,
	/* num_resources     */		XtNumber(resources),
	/* xrm_class         */		NULLQUARK,
	/* compress_motion   */		TRUE,
	/* compress_exposure */		TRUE,
	/* visible_interest  */		FALSE,
	/* destroy           */		NULL,
	/* resize            */		Resize,
	/* expose            */		Redisplay,
	/* set_values        */		SetValues,
	/* accept_focus      */		NULL,
 }
};

/* Class record pointer */
WidgetClass labelWidgetClass = (WidgetClass) &labelClassRec;

/* Private procedures */
	.
	.
	.
```

6.4. Class Initialization: the class_initialize procedure **6.4. Class Initialization: the class_initialize procedure**

Many class records can be initialized completely at compile time. But in some cases a class may want to register type converters, inherit procedures from its superclass by copying them from its superclass's record, register unique callback kinds, 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 called exactly once automatically by the toolkit. A class initialization procedure takes no parameters and returns no result:

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

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 calling the class_initialization procedure if it is FALSE. The class_inited field is then set to TRUE. After the one-time initialization, a class structure is constant.

Here is the class initialization procedure for ButtonBox, a subclass of Composite. ButtonBox inherits its superclass's insert_child and delete_child routines. That is, it uses whatever routines Composite has defined for these fields:

```
static void ClassInitialize()
{
        CompositeWidgetClass superclass;
        ButtonBoxWidgetClass myclass;

        myclass = (ButtonBoxWidgetClass) buttonBoxWidgetClass;
        superclass = (CompositeWidgetClass) myclass->core_class.superclass;

/* Inherit insert_child and delete_child from Composite */
        myclass->composite_class.insert_child =
          superclass->composite_class.insert_child;
        myclass->composite_class.delete_child =
          superclass->composite_class.delete_child;
}
```

6.5. XtClass 6.5. XtClass

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

w Specifies the widget.

XtClass returns a pointer to the widget class structure. It is simply an abbreviation for:
w->core.widget_class

6.6. XtSuperclass 6.6. XtSuperclass

WidgetClass XtSuperclass(*w*)
    Widget *w*;

w Specifies the widget.

XtSuperclass returns a pointer to the widget's superclass class structure. It is simply an abbreviation for:
w->core.widget_class->core_class.superclass

6.7. XtIsSubclass 6.7. 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 specified w has a class, which is equal to or is a 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.

Note that the Boolean function XtIsComposite is simply an abbreviation for XtIsSubclass.

XtIsSubclass(*w*, *compositeWidgetClass*)

6.8. XtCheckSubclass 6.8. XtCheckSubclass

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

parent   Specifies the widget under question.

widget_class   Specifies the widget class to test against.

XtCheckSubclass determines if the specified w has a class that is equal to or is a subclass of widget_class. If not, XtCheckSubclass issues an error and aborts the program. It should be used at the entry-point of routines that can be called by clients in order to determine 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.

Note

This routine currently is not yet implemented.

6.9. Superclass Chaining 6.9. 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. These are:

- class_name
- widget_size
- realize
- actions
- visible_interest
- resize
- expose
- accept_focus

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

- initialize
- set_values
- resources (For subclasses of Constraint, the constraints 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 on up the class chain to the Core class. The only subclass-to-superclass field is ‘‘destroy’’.

6.10. Inheriting Superclass Operations 6.10. 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

To inherit an operation, you simply write code in the class_initialize procedure to copy the superclass's procedure into the subclass structure. See the example above in ‘‘Class Initialization’’.

6.11. Calling Superclass Operations 6.11. Calling Superclass Operations

A widget class sometimes wants to explicitly call a superclass operation that is not normally 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 (for example, Scrollbar and the top-level widget) implement insert_child by first calling their superclass's insert_child procedure and then calling XtManageChild to add the child to the managed list.

Here is the code for the top-level widget's *insert_child*:

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

7. The Widget Tree 7. The Widget Tree

Widgets are either ‘‘primitive’’ or ‘‘composite’’. A primitive widget has no children widgets and can never have them. A composite widget may in unusual circumstances have zero children but usually has at least one. Primitive widgets are leaves of the widget tree. Composite widgets are intermediate nodes with children and the top-level widget returned by XtInitialize is the root of the widget tree.

Except for pop-up widget windows (which are children of the root window so that they are not clipped), the widget tree exactly duplicates the associated window tree.

The widget tree is manipulated by several toolkit functions. For example, XtRealizeWidget traverses the tree downward to recursively realize widgets. XtDestroyWidget traverses the tree downward to destroy all children. The internal routine _XtGetResources traverses 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 the widget's geometry.

To facilitate up-traversal of the widget tree, each widget has a pointer to its parent widget. (The top-level widget returned by XtInitialize has a parent pointer of NULL.)

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

8. Window Attributes 8. Window Attributes

The Core widget definition contains screen and window. Therefore, widgets and intrinsics do not have to pass them. The window field may be NULL for a while (see ‘‘Creating Widgets’’ and ‘‘XtRealizeWidget’’).

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

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

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

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

They all take a widget and return the specified value.

Several window attributes are locally cached in the widget, so they can be set by the resource manager and by XtSetValues, and can be used by routines that derive structures from these values (for example, depth for deriving pixmaps, background_pixel for deriving GCs, and so on), as well as used 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 toolkit. When an XConfigureWindow is issued on the widget's window, 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.

9. Creating Widgets 9. Creating Widgets

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

1. The widgets are allocated and initialized with resources and are optionally added to the managed subset of their parent, with whom they negotiate geometry.
2. The widgets create X windows and get mapped.

To start the first phase, the application calls XtCreateWidget on all its widgets and adds some (usually, most or all) of its widgets to their respective parents' managed set. Adding a child to its parent's managed set involves geometry negotiation. That is, the parent may force some of the child's siblings to change size and position, and the parent 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 all widgets have been created and are managed by their respective parents. Consequently, in the first phase, no X windows are actually created because chances are they would just get moved around after creation, which is slow.

After all widgets have been created, the application calls XtRealizeWidget on the top-level widget to start the second phase. XtRealizeWidget recursively traverses the widget tree, allocates an X window to each widget, and maps the widgets.

9.1. XtCreateWidget 9.1. 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 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.

XtCreateWidget performs much of the ‘‘boiler-plate’’ operations of widget creation. It

- Checks that the parent is a subclass of compositeWidgetClass
- Allocates memory for the widget instance
- Initializes the core non-resource 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
- Calls the initialize procedures for the widget, starting at the Core initialize procedure on down to the widget’s initialize procedure.
- Puts the widget into its parent’s children list by calling its parent’s insert_child procedure

(See ‘‘Adding New Children: the insert_child procedure’’.

9.1.1. Widget Initialization: the initialize procedure 9.1.1. Widget Initialization: the initialize procedure

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

The main job of an initialization procedure is to fill in fields in the new widget record. These are generally nonresource fields (for example, GCs that the widget uses) or resource fields left unspecified (for example, if width and height are zero (0), the widget fills in width and height with appropriate values based on other resources).

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.

However, 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. In this case the subclass must modify or recalculate fields declared and computed by its superclass. For example, a subclass may 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. A typical widget should place itself into whatever size is specified (if any), but it computes a reasonable size for itself 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 and is updated by all superclasses so far. A subclass initialize procedure may 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 this is so, it just adds its surround size to the width and height fields in the new widget. If this is not so, it must make do with the size originally specified. The ‘‘new’’ widget will become the instance record. Therefore, any callback lists or similar elements that the initialization procedure registers should specify ‘‘new’’ as the widget instance.

The arguments specified by args and num_args in the XtCreateWidget call are passed in to the initialize procedure. These are are ignored by most widgets. However, these are used by a few widgets that have subparts that are not widgets but, nonetheless, have resources that must be filled in. See ‘‘XtGetSubresources’’ for further information.

9.2. XtRealizeWidget 9.2. XtRealizeWidget

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

w Specifies the widget.

If the widget is already realized, XtRealizeWidget simply returns.

Otherwise, XtRealizeWidget 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 XtRealizeWidget recursively calls itself on each of the widget’s managed children. Finally, XtRealizeWidget maps all of the managed children windows that have mapped_when_managed TRUE. If num_children equals num_mapped_children, it calls XMapSubwindows to map all the children at once. Otherwise, it maps each child individually. If the widget is the special top-level widget (that is, its parent is NULL), XtRealizeWidget maps the widget window.

XtCreateWidget, XtRealizeWidget, XtAdd/RemoveChildren, 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.

9.3. Window Creation: the realize procedure 9.3. Window Creation: the realize procedure

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

w Specifies the widget.

value_mask Specifies which fields in attributes to use.

attributes Specifies the window attributes to use in the XCreateWindows call.

The realize procedure must make the window a reality.

The generic procedure XtRealizeWidget 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.
- do_not_propagate_mask is set to propagate all mouse and keyboard events up the window tree. A composite widget can implement functionality caused by 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. The realize procedure defined for Core simply calls XtCreateWindows with the passed value_mask and attributes, as well as with windowClass and visual set to CopyFromParent. Both CompositeWidgetClass and ContraintWidgetClass 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 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.

9.3.1. XtCreateWindow 9.3.1. XtCreateWindow

Rather than call the XCreateWindow X function directly, a realize procedure should call the toolkit analog XtCreateWindow. This routine simplifies the creation of windows for widgets. The definition for this function is:

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.
value_mask	Specifies which fields in attributes to use.
attributes	Specifies the window attributes to use in the XCreateWindows call.

XtCreateWindow calls XCreateWindow with values from the widget structure and the passed parameters, as well as assigns the created window into the widget’s window field.

9.4. XtIsRealized 9.4. XtIsRealized

The definition for this function is:

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

w Specifies the widget.

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

10. Destroying Widgets 10. Destroying Widgets

Destroying widgets is simple. The 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

10.1. Adding and Deleting Destroy Callbacks 10.1. Adding and Deleting Destroy Callbacks

The destroy callback uses the mechanism described in ‘‘Callbacks’’. The callback list is identified by XtCallbackKind destroyCallback. To add a destroy callback procedure ClientDestroy with closure *closure* to Widget w, call XtAddCallback. To remove the callback, call XtRemoveCallback. Both take the following parameter list:

(*w*, *destroyCallback*, *ClientDestroy*, *closure*)

10.2. XtDestroyWidget 10.2. XtDestroyWidget

To destroy a widget instance, use XtDestroyWidget. The definition for this function is:

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

w Specifies the widget.

This routine is the only method of destroying a widget, including widgets that wish to destroy themselves. It may be called at any time, including from an application callback routine of the widget being destroyed. This requires a two-phase destroy process. In phase one, XtDestroyWidget performs the following actions:

- If being_destroyed 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.

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.

In phase two, XtDestroyWidget destroy 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 ‘‘Deleting Children: the delete_child procedure’’.)
- Calls the destroy procedures for the widget (and all descendants) in post-order. For each such widget, it calls the destroy procedure for the widget, then calls the destroy procedure for its superclass, until finally it calls the Core destroy routine.
- Calls XDestroyWindow if the widget is realized (that is, has an X window) Otherwise, recursively descends the tree and calls XDestroyWindow on the top widget of each realized subtree. The server recursively destroys all descendant windows.

10.3. Deallocating Dynamic Data: the destroy procedure 10.3. Deallocating Dynamic Data: the destroy procedure

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 WidgetClass should 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 .

11. Callbacks 11. Callbacks

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 may 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.

11.1. Callback Procedure and CallbackList Definitions 11.1. Callback Procedure and CallbackList Definitions

```
typedef Opaque XtCallbackList; /* List of Callback procedures */

typedef void (*XtCallbackProc)();
    Widget w;
    caddr_t closure;
    caddr_t call_data;
```

w Specifies widget for which the callback is registered.

closure Specifies the client 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 a closure of 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 address of a struct containing the position and height of the thumb.

11.2. Registering CallbackKinds 11.2. Registering CallbackKinds

Whenever a widget contains a callback list for use by clients, it also exports in its public ‘‘.h’’ file a name of type XtCallbackKind to identify the callback list. These names are registered by the class_initialize procedure, which calls XtNewCallbackKind to get a unique identifier of the callback. The definitions for XtCallbackKind and this function are:

typedef Opaque XtCallbackKind;

XtCallbackKind XtNewCallbackKind(*widget_class*, *offset*)
    WidgetClass *widget_class*;
    Cardinal *offset*;

widget_class Specifies the widget class which exports the callback list.

offset Specifies the offset of the callback list in the widget structure. (See ‘‘XtOffset’’ for further information.)

Clients and widgets never access widget callback fields directly. Instead, they always identify the desired callback list using the exported XtCallbackKind variable. All callback manipulation routines described below check that the requested callback list is indeed implemented by the widget.

Because callback kinds are registered by the class_initialize procedure, clients should not expect the exported callback identifiers to be valid before creating a widget of that class.

11.3. XtAddCallback 11.3. XtAddCallback

To add a callback procedure to a callback list, use XtAddCallback. The definition for this function is:

void XtAddCallback(*w*, *callback_kind*, *callback*, *closure*)
    Widget *w*;
    XtCallbackKind *callback_kind*;
    XtCallbackProc *callback*;
    caddr_t *closure*;

w Specifies the widget to add the callback to.

callback_kind Specifies the list of callbacks within the widget to append to.

callback Specifies the callback procedure to add.

closure Specifies the closure to be passed to the callback when it is invoked by XtCallCallbacks (which may often be NULL).

11.4. XtRemoveCallback 11.4. XtRemoveCallback

To remove a callback procedure from a callback list, use XtRemoveCallback. The definition for this function is:

```
void XtRemoveCallback(w, callback_kind, callback, closure)
    Widget w;
    XtCallbackKind callback_kind;
    XtCallbackProc callback;
    caddr_t closure;
```

w Specifies the widget to delete the callback from.

callback_kind Specifies the list of callbacks within the widget to remove the callback from.

callback Specifies the callback procedure to delete.

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

11.5. XtRemoveAllCallbacks 11.5. XtRemoveAllCallbacks

To remove all callback procedures from a callback list (and, thus, free all storage associated with the callback list), use XtRemoveAllCallbacks. The definition for this function is:

```
void XtRemoveAllCallbacks(w, callback_kind)
    Widget w;
    XtCallbackKind callback_kind;
```

w Specifies the widget to remove the callback from.

callbacks Specifies the callback list within the widget to remove.

11.6. XtCallCallbacks 11.6. XtCallCallbacks

To execute the procedures in a callback list, use XtCallCallback. The definition for this function is:

```
void XtCallCallbacks(w, callback_kind, call_data)
    Widget w;
    XtCallbackKind callback_kind;
    caddr_t call_data;
```

w Specifies the widget to execute the callbacks of.

callback_kind Specifies the list of callbacks 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) word is needed; the call_data is the address of the data if more than one word is needed (for example, Scrollbar needs to pass back the height and position of the scroll thumb).

12. Mapping Strings to Widget Instances and Classes 12. Mapping Strings to Widget Instances and Classes

12.1. XtStringToWidget 12.1. XtStringToWidget

XtStringToWidget maps a fully qualified widget name to a widget instance. The definition for this function is:

Widget XtStringToWidget(*name*);
    String *name*;

name The fully qualified name of the desired widget.

Note

This routine currently is not yet implemented.

12.2. XtStringToClass 12.2. XtStringToClass

XtStringToClass maps a fully qualified widget name to a widget class. The definition for this function is:

WidgetClass XtStringToClass(*name*);
    String *name*;

name The name of the desired widget class.

Note

This routine currently is not yet implemented.

13. Composite Widgets 13. Composite Widgets

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 generic 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 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, an emacs-like composite editor widget might allocate one child widget per 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 XtManageChildren and XtUnmanageChildren, which notify the parent to recalculate the physical layout of its children by calling the parent's change_managed procedure.

Most managed children are mapped, but some widgets may be in a state where they take up physical space but do not show anything. Managed widgets will not be mapped 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 three classes:

- Fixed boxes have a fixed number of children that are created by the parent. All of these children are managed, and none of them ever make geometry manager requests. An example is Scrollbar.
- Homogeneous boxes treat all children equally and apply the same geometry constraints to each child. Many clients insert and delete widgets freely (for instance, xmh's folder box). Examples are Menu, ButtonBox, and Pane.
- 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 in terms of the relationship between a child and the parent's borders as well as between the child and other, specific children. Most clients insert a fixed number of widgets into a homogeneous box. Heterogeneous boxes are usually subclasses of Constraint. An example is Form.

Each composite widget, especially those that are heterogeneous, may 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.

13.1. Composite Widget Type Definitions 13.1. Composite Widget Type Definitions

typedef Widget **WidgetList*;

13.2. XtIsComposite 13.2. XtIsComposite

To test if a widget is a subclass of Composite, use XtIsComposite. The definition for this function is:

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

w Specifies the widget under question.

XtIsComposite is just an abbreviation for XtIsSubclass with the following parameters list:

(*w, compositeWidgetClass*).

13.3. Adding New Children: the insert_child procedure 13.3. Adding New Children: the insert_child procedure

To add the child to the parent's children array, XtCreateWidget calls the parent's class routine insert_child.

Fixed boxes usually insert the child first by calling Composite's insert_child routine, then by calling XtManageChild to add the child to the box's managed set. Homogeneous boxes usually just inherit Composite's insert_child routine. Heterogeneous boxes usually have a separate array of constraint information on the side and, so, perform the insertion with custom code.

An insertion procedure takes the widget to insert and the argument list used to create the widget. This argument list is usually used only by homogeneous boxes (subclasses of Constraint).

```
typedef void (*XtArgsProc)();
    Widget w;
    ArgList args;
    Cardinal num_args;
```

w Specifies the widget to insert.

args Specifies the argument list passed to XtCreateWidget.

num_args Specifies the number of arguments in argument list.

13.3.1. Insertion Order 13.3.1. Insertion Order

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

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. 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.

Using zero (0) means before all other children, while using 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.

13.4. Deleting Children: the delete_child procedure 13.4. Deleting New Children: the delete_child procedure

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.

Fixed boxes usually do not need a delete_children routine because they create their own children. Hence, they need not worry about a client calling XtDestroyWidget on one of their children. Homogeneous boxes usually just inherit Composite's delete_child routine. Heterogeneous boxes usually have a separate array of constraint information on the side and, so, perform the deletion with custom code.

A deletion procedure is of type XtWidgetProc, and it merely takes the widget to delete.

13.5. Adding and Removing Children from the Managed Set 13.5. Adding and Removing Children from the Managed Set

The X Toolkit provides a set of generic routines to permit, without explicit knowledge of the composite widget type. 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.

13.5.1. XtManageChildren 13.5.1. XtManageChildren

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. (Note that some composite widgets, especially fixed boxes, call XtManageChild from their insert_child procedure.) The definition for XtManageChildren is:

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 actions:

- Issues an error if the children do not all have the same parent.
- Returns immediately if the common parent is being destroyed.
- Otherwise, it processes the unique, unmanaged children:
  – A 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.
- Calls the change_managed routine of the widgets' parent.
- If the parent is realized, XtManageChildren makes (some of) the newly managed children visible:
  – Calls XtRealizeWidget on each previously unmanaged child that is unrealized.
  – Maps each previously unmanaged child that has map_when_managed TRUE.

The composite widget's change_managed procedure is called to notify the parent that its set of managed children has changed. It may, as it sees fit, reposition and resize any of its children. It moves each child as needed by using the XtMoveWidget procedure, which first updates the x and y fields. Then, it 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, which first updates the width, height, and border_width fields. Then, it calls XConfigureWindow if the widget is realized.

13.5.2. XtManageChild 13.5.2. XtManageChild

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. The definition for this function is:

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

child Specifies the child to add.

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

13.5.3. XtUnmanageChildren 13.5.3. XtUnmanageChildren

To remove a list of children from the managed list of their parent, use XtUnmanageChildren. The definition for this function is:

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, it processes the unique, managed children:
  - A child is ignored if it is already unmanaged or being destroyed.
  - Otherwise, XtUnmanagedChildren marks the child as unmanaged.
  - If the parent is realized, makes child 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.

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 entirely destroy widgets, see ''XtDestroyWidget''.

13.5.4. XtUnmanageChild 13.5.4. XtUnmanageChild

To remove a single child from its parent's managed set, use XtUnmanageChild. The definition for this function is:

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

child Specifies the child to remove.

XtUnmanageChild constructs a WidgetList of length one (1) and calls XtUnmanageChildren.

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

13.6. XtSetMappedWhenManaged 13.6. XtSetMappedWhenManaged

A widget is normally mapped if it is managed. This behavior can be overridden by setting the map_when_managed field FALSE. To change the map_when_managed field, use XtSetMappedWhenManaged. The definition for this function is:

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

w Specifies the widget.

map_when_managed Specifies the new value.

If the widget is realized and managed, XtSetMappedWhenManaged maps or unmaps the window as appropriate.

14. Toolkit Initialization 14. Toolkit Initialization

To initialize the X Toolkit, call the XtInitialize function. The definition of the functions is;

Widget XtInitialize(*name*, *class_name*, *options*, *num_options*, *argc*, *argv*)
    String *name*;
    String *class_name*;
    XrmOptionDescRec *options*;
    Cardinal *num_options*;
    Cardinal **argc*;
    char **argv*[];

name	Name of this instance of the application.
class	Class name of this application. Usually the generic name for all instances of this application.
options	Specifies how to parse the command line for any application-specific resources. The options is passed as a parameter to XrmParseCommand. (See the Xlib manual for further information.)
num_options	The number of entries in options list.
argc	Address of the number of command line parameters.
argv	Command line parameters.

XtInitialize must be called before calling any other X Toolkit functions. It parses the command line, opens the display, and initializes the toolkit. It returns the ''top-level'' widget to use as the parent of the application's root widget.

It 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.

14.1. Setting Resources from the Command Line 14.1. Setting Resources from the Commmand Line

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 described in the Xlib documentation.

The standard table contains the following entries:

Option string	Resource name	Argument Kind	Resource value
=	geometry	IsArg	
-bd	border	SepArg	
-bordercolor	border	SepArg	
-bg	background	SepArg	
-background	background	SepArg	
-bw	borderWidth	SepArg	
-border	borderWidth	SepArg	
-fg	foreground	SepArg	
-foreground	foreground	SepArg	
-fn	font	SepArg	
-font	font	SepArg	
-rv	reverseVideo	NoArg	on [not implemented]
-reverse	reverseVideo	NoArg	on [not implemented]

+rv	reverseVideo	NoArg	off [not implemented]
-n	name	SepArg	
-name	name	SepArg	
-title	title	SepArg	
-t	title	SepArg	

15. Memory Management 15. 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, 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. The definitions for the X Toolkit memory management routines are:

char *XtMalloc(*size*);
    Cardinal *size*;

size Number of bytes desired.

XtMalloc returns a pointer to a block of storage of at least the specified size bytes.

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

num Number of array elements to allocate.

size Size in bytes of an array element.

XtCalloc allocates space for the specified number of array elements of the specified size bytes.

char *XtRealloc(*ptr*, *num*);
    char **ptr*;
    Cardinal *num*;

ptr Pointer to old storage.

num 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 into the new block (as much as will fit).

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

ptr Pointer to block of storage to free.

Free returns storage and allows it to be reused.

16. Input and Events 16. Input and Events

While X allows the reading and processing of events anywhere in an application, widgets in the X Toolkit neither read events nor grab the server or mouse. Widgets merely register procedures 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 events 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 window 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.
- Disable the dispatching of user-initiated events (keyboard and mouse events) for a particular widget.
- Constrain the dispatching of events to a cascade of ‘‘pop-up’’ widgets.
- Call the appropriate set of procedures currently registered when an event is read.

In general, widgets do not use any of the event manager routines. That is, the normal interface to X events is through the higher-level Translation Manager, which maps sequences of X events (with modifiers) into procedure calls. In general, applications do not use any of the event manager routines, except XtMainLoop.

16.1. Adding and Deleting Event Sources 16.1. Adding and Deleting 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 timer events and file data pending events into this mechanism.

The next two functions provide input gathering from files. The application registers the files with the read routine. When input is pending on one of the files, a message event containing the window that is to receive the input is generated.

16.1.1. XtAddInput 16.1.1. XtAddInput

To register a new file for input, use XtAddInput. The definition for this function is:

```
void XtAddInput(w, source, condition)
    Widget w;
    int source;
    caddr_t condition;
```

w Specifies the widget.

source Specifies the source file descriptor in UNIX or other operating system dependent device specification.

condition Specifies the mask that indicates a read, write, or exception condition.

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, a ClientMessage event that contains the window, the source, and the condition is returned to the application.

16.1.2. XtRemoveInput 16.1.2. XtRemoveInput

To discontinue a source of input, use XtRemoveInput. The definition for this function is:

void XtRemoveInput(*w*, *source*, *condition*)
    Widget *w*;
    int *source*;
    caddr_t *condition*;

w Specifies the widget.

source Specifies the source file descriptor in UNIX or other operating system dependent device specification.

condition Specifies the mask that indicates a read, write, or exception condition.

The XtRemoveInput function causes the X read routine to stop gathering input from the input source. The condition parameter allows the disabling of input only for certain conditions.

16.1.3. XtAddTimeOut 16.1.3. XtAddTimeOut

The timeout facility notifies the application or the widget writer through a message event that a specified time interval has elapsed. Timeout values are set for a single widget and are uniquely identified by an interval ID. This allows for more than one widget to specify a timeout and for a single widget to have multiple timeouts.

To create a timeout value for a widget, use XtAddTimeOut. The definition for this function is:

XtIntervalId XtAddTimeOut(*w*, *interval*)
    Widget *w*;
    unsigned long *interval*;

w Specifies the widget.

interval Specifies the time interval in milliseconds.

The XtAddTimeOut function creates a timeout value for the widget that is identified by the widget ID and returns an identifier for that timer. The timeout value is set to interval. This causes a message event to be generated for the specified widget window whenever the time interval elapses.

16.1.4. XtRemoveTimeOut 16.1.4. XtRemoveTimeOut

To clear a timeout value, use XtRemoveTimeOut. The definition for this function is:

void XtRemoveTimeOut(*timer*)
    XtIntervalId *timer*;

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

XtRemoveTimeOut removes the timeout, even if it has triggered but has not been fetched with XtNextEvent.

16.2. Filtering X Events 16.2. 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.

16.2.1. Mouse Motion Compression 16.2.1. Mouse Motion Compression

Widgets may have a hard time keeping up with mouse 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. All but the last motion event is ignored in a consecutive sequence of motion events queued for the widget's window, and only the most recent motion event is delivered.

16.2.2. XtSetSensitive 16.2.2. XtSetSensitive

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

w Specifies the widget.

sensitive Specifies whether or not the widget should receive keyboard and mouse 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 any more.

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 may be insensitive because its sensitive field is FALSE or one of its parents is insensitive, and, thus, the widget's ancestor_sensitive field also is FALSE. A widget may but does not nned 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 ancestor_sensitive are 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, the widget's set_values procedure should take whatever display actions are needed, such as greying out or stippling the widget, when one of these changes.

16.3. Querying Event Sources 16.3. 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.

16.3.1. XtPending 16.3.1. XtPending

To determine if there are any events on the input queue, use XtPending. The definition for this function is:

```
Boolean XtPending()
```

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

16.3.2. XtPeekEvent 16.3.2. XtPeekEvent

To flush the X output buffer and return the value from the head of the input queue without removing input from the queue, use XtPeekEvent. The definition for this function is:

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

event Returns the event information to the specified event structure.

If no X input is on the queue, XtPeekEvent looks at all other input sources and returns the first source that has input pending. If no input is pending, however, XtPeekEvent blocks until input is available.

16.3.3. XtNextEvent 16.3.3. XtNextEvent

To flush the X output buffer and return the value from the head of the input queue, use XtNextEvent. The definition for this function is:

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

event Returns the event information to the specified event structure.

If no input is on the X input queue, XtNextEvent looks at the other input sources or timeout values and returns input generated by them. If no input of any sort is available, XtNextEvent blocks until an event can be returned.

Because the toolkit event processing gives preference to X events over source and timeout events, a call to XtPeekEvent followed by a call to XtNextEvent may not return the same event if XtPeekEvent returns a source or timeout event.

16.4. Adding and Removing Event Handlers 16.4. Adding and Removing Event Handlers

Event handlers are procedures that are called when a specified set of events occurs in a widget.

```
typedef void (*XtEventHandler());
    Widget w;
    caddr_t closure;
    XEvent *event;
```

w The widget that this event handler was registered with.

closure Client specific information registered with the event handler, this is usually NULL if the event handler is registered by the widget itself.

event The triggering event.

16.4.1. XtAddEventHandler 16.4.1. XtAddEventHandler

To register an event handler procedure with the dispatch mechanism, use XtAddEventHandler. The definition for this function is:

void XtAddEventHandler(*w*, *event_mask*, *nonmaskable*, *proc*, *closure*)
    Widget *w*;
    XtEventMask *event_mask*;
    Boolean *nonmaskable*;
    XtEventHandler *proc*;
    caddr_t *closure*;

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

event_mask Specifies the event mask to call this procedure for.

nonmaskable Specifies whether this procedure should be called on the nonmaskable events. These are event of type GraphicsExpose, NoExpose, CreateNotify, MapRequest, ConfigureRequest, ResizeRequest, CirculateRequest, SelectionClear, SelectionRequest, ClientMessage, and MappingNotify.

proc Specifies the client event handler procedure.

closure 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 OR'ed into the existing mask. If the widget is realized XtAddEventHandler calls XSelectInput if necessary.

16.4.2. XtRemoveEventHandler 16.4.2. XtRemoveEventHandler

To remove a previously registered event handler, use XtRemoveEventHandler. The definition for this function is:

void XtRemoveEventHandler(*w*, *event_mask*, *nonmaskable, proc*, *closure*)
    Widget *w*;
    XtEventMask *event_mask*;
    Boolean *nonmaskable*;
    XtEventHandler *proc*;
    caddr_t *closure*;

w Specifies the widget for which this procedure is registered.

event_mask

nonmaskable Specifies the events to unregister this procedure for

proc Specifies the event handler procedure registered.

closure Specifies the closure 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 AllEvents and with nonmaskable TRUE.

16.5. Constraining Events to a Cascade of Widgets 16.5. 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’’, user events (keyboard and mouse 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 widgets outside of the modal widgets. Menus may pop-up submenus and dialog boxes may 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 modal widget in the cascade are if they occur outside the cascade are: KeyPress, KeyRelease, ButtonPress, and ButtonRelease.

Events that are ignored if they occur outside the cascade are: MotionNotify, EnterNotify, LeaveNotify, FocusIn, FocusOut, KeymapNotify, PropertyNotify, SelectionClear, SelectionRequest, ColormapNotify, ClientMessage, and MappingNotify. All other event are delivered normally.

Modal widgets use the procedures XtAddGrab and XtRemoveGrab to constrain user events to a modal cascade and to subsequently remove a grab when the modal widget goes away.

16.5.1. XtAddGrab 16.5.1. XtAddGrab

The definition for this function is:

void XtAddGrab(*w*, *exclusive*)
    Widget *w*;
    Boolean *exclusive*;

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.

XtAddGrab appends the widget to the modal cascade.

When XtDispatchEvent tries to dispatch a user event and when at least one modal widget is in the widget cascade, it first determines if the 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 exclusive TRUE.

This subset of the modal cascade is the active subset. User events that occur outside the widgets in this subset are ignored. Modal menus 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, ButtonRelease events are always delivered to the most recent widget in the cascade.

Note

This routine currently is not yet implemented.

16.5.2. XtRemoveGrab 16.5.2. XtRemoveGrab

The definition for this function is:

```
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.

Note

This routine currently is not yet implemented.

16.6. Dispatching Events 16.6. 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 receeved, 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 (a ‘‘closure’’). 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.

16.6.1. XtDispatchEvent 16.6.1. XtDispatchEvent

To send events to registered functions and widgets, use XtDispatchEvent. Normally, this procedure is not called by client applications (see ‘‘XtMainLoop’’). The definition for this function is:

```
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 can also be used to dispatch user-constructed events.

16.6.2. XtMainLoop 16.6.2. XtMainLoop

Applications normally process input within the X Toolkit by calling the XtMainLoop routine.

The definition for this function is:

```
void XtMainLoop()
```

XtMainLoop reads the next incoming file, timer, or X event. Then, it dispatches this to the appropriate registered procedure. This is the main loop of X Toolkit applications, and, as such, it does not return. Applications are expected to exit directly in response to some user action.

17. Widget Visibility and Exposure 17. Widget Visibility and Exposure

Every primitive widget (and some composite widgets) displays data on the screen by means of raw X calls. Widgets cannot simply write to the screen and forget what they've done. That is, widgets must keep enough state to redisplay the window (or parts of it) if a portion is obscured and then re-exposed.

17.1. Widget Redisplay: the expose procedure 17.1. Widget Redisplay: the expose procedure

typedef void (**XtExposeProc*)();
  Widget *w*;
  XEvent **event*;

Redisplay of a widget upon exposure is the responsibility of the expose procedure in the widget's class record. A small simple widget (for example, Label) may ignore the bounding box information in the Expose event and just redisplay the entire window. A more complicated widget (for example, Text) may use the bounding box information to minimize the amount of calculation and redisplay it does. The expose procedure is responsible for exposure of all superclass data as well as its own, 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. See ‘‘Inheriting Superclass Operations’’.

17.2. Exposure Compression 17.2. Exposure Compression

Many widgets are so simple that they always redisplay the entire widget in their expose procedure. These widgets do not care about getting partial expose events. If the compress_expose field in the widget class structure is TRUE, the Event Manager calls the widget's expose procedure only for Expose events in which the expose count is zero (0).

17.3. Widget Visibility 17.3. 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 may 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 ExposeNotify 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.

18. Size and Position of Widgets Geometry Management 18. Size and Position of Widgets Geometry Management

A widget does not directly control its size and location. This is the responsibility of the parent of that widget. In general, the position of children is left up to their parent. However, the widget itself often has the best idea of its optimal size and may also have a preferred location.

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 depth of the widget's children. The only exception are fixed boxes like Scrollbar, 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 can not use X calls directly. Instead, they must ask their parent's geometry manager to change it for them. 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.

When the geometry manager is asked to change the geometry of a particular child, the geometry manager may also rearrange and resize any or all of the other children 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), it should inform each affected child 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 may involve extended negotiation, windows are not actually allocated to widgets at application startup until all widgets are satisfied with their geometry. See ''XtRealizeWidget'' and ''Creating Widgets'' for more details.

18.1. Making Geometry Manager Requests 18.1. Making Geometry Manager Requests

To make a general geometry manager request, use XtMakeGeometryRequest. The definition for this function is:

XtGeometryResult XtMakeGeometryRequest(*w*, *request*,*reply*)
    Widget *w*;
    XtWidgetGeometry **request*;
    XtWidgetGeometry **reply*;

w	Specifies the widget.
request	Specifies the desired widget geometry (size, position, border width, and stacking order).
reply	Specifies an argument in which the allowed widget size is returned. 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,
} XtGeometryResult;
```

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

```
typedef unsigned long XtGeometryMask;

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

The request_mask definitions are from <X/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_mask to determine which fields in the XtWidgetGeometry structure you want to specify.

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

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

(See the Xlib manual for definition and behavior of Above , Below , TopIf , BottomIf , and Opposite .)

The XtMakeGeometryRequest function is a simple routine. If the parent is not a subclass of Composite, XtMakeGeometryRequest issues an error. If the parent's geometry_manager is NULL, XtMakeGeometryRequest issues an error. If being_destroyed is TRUE, XtMakeGeometryRequest returns XtGeometryNo . If managed is FALSE, XtMakeGeometryRequest issues a warning and returns XtGeometryNo . Otherwise, XtMakeGeometryRequest calls the parent's geometry_manager procedure with the given parameters.

If the parent's geometry manager returns XtGeometryYes and the widget is realized, XtMakeGeometryRequest reconfigures the widget's window by calling XResizeWindow .

18.2. Making Resize Requests 18.2. Making Resize Requests

To make a simple resize request, use XtMakeResizeRequest. The definition for this function is:

```
XtGeometryResult XtMakeResizeRequest(w, width, height, replyWidth, replyHeight)
    Widget w;
    Dimension width, height;
    Dimension *replyWidth, *replyHeight
```

w Specifies the widget.

width Specifies the desired widget width.

height Specifies the desired widget height.

replyWidth Specifies an argument in which the allowed widget width is returned.

replyHeight Specifies an argument in which the allowed widget height is returned.

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 XtResizeRequest requesting the compromise width and height.

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

18.3. Geometry Manager Responsibilities: the geometry_manager procedure 18.3. Geometry Manager Responsibilities: the geometry_manager procedure

Specific geometry managers are of type XtGeometryHandler:

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

A zero (0) bit in the request mask means that you do 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 you want 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 position fields (x, y) with XtMoveWidget, and updates width, height, and border_width directly, and then returns XtGeometryYes. The value of the reply parameter is undefined. (The widget’s window is resized automatically by XtMakeGeometryRequest.)

Note that the geometry manager does not call the widget’s resize procedure if it returns XtGeometryYes. Then, the widget must perform whatever resizing calculations are needed explicitly. This costs primitive widgets an extra line, but it substantially simplifies the writing of nice composite widgets.

If the geometry manager chooses to disallow the request, the widget cannot change it geometry. The value of the reply 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), satisfy a request only if other changes take place (for example, size can be changed if the widget is moved to a particular location), or satisfy a lesser request (for example, it can't 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 reply with the actual changes it is willing to make and returns XtGeometryAlmost. If a bit in reply.request_mask is zero (0), the geometry manager will not change the corresponding value. If a bit is one (1), the geometry manager will change that element to the corresponding value in the reply parameter.

When XtGeometryAlmost is returned, the widget must decide if the compromise suggested in the reply is acceptable. If so, the widget must not change its geometry directly. Rather, it should make another call to XtMakeGeometryRequest.

If the next geometry request on this window uses the reply box filled in by an XtGeometryAlmost return and if there have been no intervening geometry requests on either its parent or any of its children, the request is guaranteed to succeed. That is, if you ask again right away with the returned geometry, you 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 may sometimes change the size of other managed children. To do this, they should call the procedure XtResizeWidget described below.

18.3.1. XtMoveWidget 18.3.1. XtMoveWidget

Geometry managers should call XtMoveWidget to move their children widget around. The definition for this function is:

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.

18.3.2. XtResizeWidget 18.3.2. XtResizeWidget

Geometry managers should call XtResizeWidget to resize children widgets (except the child making the geometry request). The definition for this function is:

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 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.

18.4. Widget Resizing Responsibilities: the resize procedure 18.4. Widget Resizing Responsibilities: the resize procedure

A child can be involuntarily resized by its parent at any time. Widgets usually want to know when they have changed size, so they can re-layout their displayed data to match the new size. When a geometry manager resizes a child it calls XtResizeWidget, which updates the geometry fields in the widget, configures the window if the widget is realized, and calls the child's resize procedure to notify the child.

The resize procedure 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 cannot issue an XtMakeGeometryRequest call from its resize procedure.

19. Input Focus 19. Input Focus

This section discusses the underlying philosophy of the X Toolkit input focus management.

Widgets that want the input focus can call XSetInputFocus directly. To allow others 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 X11 focus notification mechanism directly. Widgets that never want the input focus should set their accept_focus procedure to NULL.

Composite widgets are required to export two additional functions: move_focus_to_next and move_focus_to_prev. These procedures move the input 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 input 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.

20. Selections 20. 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. See the Xlib manual for details.

21. Resource Management 21. Resource Management

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, 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 file (by using either the name or class), and by XtSetValues. In addition, it is readable by XtGetValues.

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

21.1. XtOffset 21.1. XtOffset

To determine the byte offset of a field within a structure, use the macro XtOffset. The definition for this macro is:

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

pointer_type Specifies a type that points to the structure.

field_name Specifies the name of the field to calculate the byte offset from.

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

21.2. XtNumber 21.2. XtNumber

To determine the number of elements in a fixed-size array, use the macro XtNumber. The definition for this macro is:

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.

21.3. Resource Lists 21.3. Resource Lists

A resource entry specifies a field in the widget, the textual name and class of the field that argument lists and external default file use to refer to the field, and a default value that the field should get if no value is specified. The declaration for the XtResource structure is:

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

The resource_name field contains the name used by clients to access the field in the widget. By convention, it starts with a 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.

The resource_class field contains the class of the resource. A resource is rarely a widget. Therefore, a resource class is not the same as a widget class, but there are some resemblances. A resource class offers two functions:

- It isolates you from different representations that widgets may use for a similar resource
- It lets you specify 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 may have several resource pixels: background, foreground, border, block cursor, mouse cursor, and so on. Typically, the background defaults to ‘‘white’’ and everything else to ‘‘black’’. The resource class for each of these resources 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, giving background a resource class of Background and all the other pixel entries a resource class of Foreground is the right thing to do. (By convention, resource classes are always spelled starting with a capital letter.)

The resource_type field is the physical representation type of the resource. By convention, it starts with an upper-case letter and is usually 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) to the desired physical representation (see ‘‘Resource Conversions’’).

The resource_size field is the size of the physical representation in bytes. The resource_offset is the offset in bytes of the field within the widget. The macro XtOffset in Intrinsic.h 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.

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. See sections ‘‘XtGetValues’’ and ‘‘XtSetValues’’.

21.4. Superclass to Subclass Chaining of Resource Lists 21.4. 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 on down to the resources specified for this widget’s class.

In general, if a widget 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. The translations field is frequently overridden by subclasses.

21.5. XtGetSubresources 21.5. XtGetSubresources

A widget needs to do nothing to get its resources. That is, XtCreateWidget does this automatically before calling the initialize procedure. Some widgets have subparts which are not children, but for which the widget would like to fetch resources. These widgets call XtGetSubresources to accomplish this. The definition for this function is:

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 *resources*.

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 need not be NULL.

XtGetSubresources constructs a name/class list from the widget (and all ancestors) and then appends the name/class pair passed in. The resources are fetched from the argument list, database, or default in the resource list and are copied into the subpart record.

21.6. Resource Conversions 21.6. Resource Conversions

The X Toolkit defines all the resources and representations used in the Core widget. It also registers some resource conversions in addition to the standard resource conversions described in the Xlib manual.

22. Reading and Writing Widget State 22. 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.

22.1. XtGetValues 22.1. XtGetValues

To retrieve the current value of a resource associated with a widget instance, use XtGetValues. The definition for this function is:

```
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 which contain the resource name and the address to store the resource value into. The arguments and values passed in args are dependent on the widget. The storage for argument values which 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.

XtGetValues starts with the resources specified for the core widget fields and proceeds down the subclass chain to the widget.

22.2. XtSetValues 22.2. XtSetValues

To modify the current value of an argument associated with a widget instance, use XtSetValues. The definition for this function is:

```
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 which contain the arguments to be modified and their new values. The arguments and values passed are dependent on the widget being modified.

num_args Specifies the number of arguments 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.

22.2.1. Setting Widget State: the set_values procedure 22.2.1. Setting Widget State: the set_values procedure

```
typedef Boolean (*XtSetValuesProc)();
    Widget current;
    Widget request;
    Widget new;
    Boolean last;
```

current Specifies the existing widget.

request Specifies the widget asked for by the XtSetValues call.

new Specifies a widget with the new values that are actually allowed.

last Specifies if this is the last set_values procedure in the chain called by XtSetValues.

The set_values procedure should construct any composite resources that depend upon changed values (for example, many GCs depend upon foreground and background). Like the initialize procedure, set_values mostly deals only with the fields defined in the subclass, but it has to resolve conflicts with superclass, especially conflicts over width and height. In this case, though, the ‘‘reference’’ widget is ‘‘current’’, not ‘‘request’’.

A ‘‘new’’ starts with the values of ‘‘request’’ and can be modified by the superclass. 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 class SetValuesProcs have been called.

If last is TRUE, the set_values procedure knows that it is last in the ‘‘superclass-to-subclass chain’’. In this case, it should perform any final actions. In particular, it should make a geometry request, if needed.

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 a redisplay. 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.

22.2.2. XtSetValuesGeometryRequest 22.2.2. XtSetValuesGeometryRequest

To make a geometry request from a set_values procedure, you can use the procedure XtSetValuesGeometryRequest. The definition for this function is:

```
XtGeometryResult XtSetValuesGeometryRequest(current, new, reply)
    Widget current;
    Widget new;
    XtWidgetGeometry *reply;
```

w Specifies the widget as it exists.

new Specifies the widget as it should be.

reply Specifies the geometry allowed if result is XtGeometryAlmost.

XtSetValuesGeometryRequest makes a geometry request if one of the geometry fields in current is different from the fields in new. (If no geometry request is needed is just returns XtGeometryNo.)

Note that only the current widget values will be updated. Do not worry about this. The XtSetValues procedure knows that it should never just overwrite the widget's geometry fields.

Note

This routine currently is not yet implemented.

23. Translation of X Events to Program/Widget functionality 23. Translation of X Events to Program/Widget functionality

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 Foo when key ‘‘y’’ is pressed.

The translation manager uses two 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 instance structure, specifies the mapping of event sequence to procedure names.

For example, the action table for the Command widget class is defined as:

```
XtActionsRec actionTable[] = {
        {"Set",          (caddr_t) Set},
        {"Unset",        (caddr_t) Unset},
        {"Highlight",    (caddr_t) Highlight},
        {"Unhighlight",  (caddr_t) Unhighlight}
        {"Notify",       (caddr_t) Notify},
};
```

The default behavior of Command is:

- Higlighted on enter window
- Unhighlighted on exit window
- Invert on left button down
- Call callbacks and reinvert on left button up

Command's default translation table is:

```
static char *defaultTranslations[] = {
        "<EnterWindow>:Highlight()",
        "<LeaveWindow>:Unhighlight()",
        "<Btn1Down>: Set()",
        "<Btn1Up>:      Notify() Unset()",
        NULL
};
```

For details on the syntax of translation tables, see Appendix B.

XtCreateWidget calls an internal routine to compile the translation specification into a structure that efficiently maps incoming events into the specified procedure calls and registers the necessary events with the Event Manager. Widgets need do nothing other than specify the action and translation tables.

23.0.1. Translation Tables 23.0.1. Translation Tables

Mapping the X Event to procedure calls is accomplished by specifying a Translation Table. Translation tables exist in textual (external) and compiled (internal) forms. The compiled form is private to the translation manager and contains the state tables, the current state, and actions to perform for each state. The textual form's syntax is described in Appendix B.

23.0.2. Function table 23.0.2. Function table

Clients of the translation manager must provide a table (array) of procedure names to procedure implementations using the following format.

```
typedef struct _XtActionsRec {
        char *string;
        caddr_t value;
} XtActionsRec, *XtActionList;
```

The string field is the name of the exported function. The value field is a pointer to any client supplied data. The most common use of this is to supply a procedure adress for the name, but Actions can map onto strings, chars, procedures, atoms, or numbers.

Each widget class has a table of name-to-procedure bindings for that class. In addition, the translation manager provides a global name to function table containing the definition of standard utility procedures for setting timers, and the current values of global values like the double-click time.

24. Graphics Contexts 24. Graphics Contexts

The X Toolkit provides a mechanism whereby cooperating clients can share Graphics Contexts, thus, reducing both the number of Graphics Contexts created and the total number of server calls in any give 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. The definition of this function is:

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.

When a Graphics Context is no longer needed it should be deallocated using the XtDestroyGC procedure. The definition for this procedure is:

```
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.

25. Errors 25. Errors

The X Toolkit uses the Xlib error handling interface. This facility allows a client to register a procedure to be called whenever an error occurs. This facility is intended for error logging but not for error correction or recovery.

Appendix A: .Xdefaults file format 25. Appendix A: .Xdefaults file format

The .Xdefaults file is an ASCII text file that consists of a number of lines with the following EBNF syntax:

```
Xdefault        = {line ’\n’}.
line            = (comment | include | production).
comment         = "!" string.
include         = "#INCLUDE" filename.
production      = [hostspec] resourcename ":" string.
resourcename    = ["*"] name {("." | "*") name}.
string          = {<any character not including eol>}.
filename        = <operating system specific>.
hostspec        = <operating system specific, same as X host spec>.
name            = {"A"-"Z" | "a"-"z" | "0"-"9"}.
```

The filename is an operating system specific name of a file to include textually in place of the include. The hostspec is a system dependent host specification. The intent is to omit any lines containing a hostspec not matching the host the application is running on. This is determined by parsing the display_name field of the display variable. Because hostspecs are likely to include colon characters (‘‘:’’) to avoid ambiguity, the hostspec must not be followed by whitespace. However, the colon that separates the resource name from the resource value string must be followed by whitespace.

Appendix B: Translation table file syntax 25. Appendix B: Translation table file syntax

A translation table file is an ASCII text file.

Notation 25. Notation

Syntax is specified in an extended BNF notation with the following abbreviations:

l ::= [a] .

is equivalent to

l ::= a | NIL .

l ::= { a } .

is equivalent to

l ::= a l | NIL .

All terminals are enclosed in ’single’ quotation marks. Informal descriptions are enclosed in <angle> brackets.

Syntax 25. Syntax

The syntax of the translation table file is:

```
translationTable   ::= { production ’\n’ }
production         ::= lhs’:’ rhs
lhs                ::= ( event | keyseq ) [ifstate] [setstate]
keyseq             ::= ’"’ keychar {keychar} ’"’
keychar            ::= [ ’^’ | ’$’ ] <ascii character>
event              ::= mode’<’eventtype’>’detail
ifstate            ::= ’,’ ’IF’ state
setstate           ::= ’,’ ’SET’ state
mode               ::= <see ‘‘Modes’’>
eventtype          ::= <see ‘‘Event Types’’>
detail             ::= <eventtype specific>
rhs                ::= { action ’(’ params ’)’ }
action             ::=  atom
params             ::= { atom | "string" | ‘char’ | number }
atom               ::= atomchar { atomchar }
atomchar           ::= { alpha | digit | ’$’ | ’_’ }
alpha              ::= <alphabetic character>
digit              ::= <decimal digit 0-9>
number             ::= digit { digit }
```

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 atoms, strings, chars, and numbers are parameters to the function.

Modes 25. Modes

The Mode field is used to specify normal X keyboard and button modifier mask bits:

Mode	Meaning
c	Control Key
Ctrl	Control Key
s	Shift Key
Shift	Shift Key

Mode	Meaning
m	Meta Key
Meta	Meta Key
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

The character ˜ when used as a prefix to the Mode specifies that the modifier key must be up. If a modifier key is omitted from a mode field specification, then it is assumed that the user does not care whether the key is up or down.

Event Types 25. 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
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

Type	Meaning
CircReq	CirculateRequest
Prop	PropertyNotify
SelClr	SelectionClear
SelReq	SelectionRequest
Select	SelectionNotify
Clrmap	ColormapNotify
Message	ClientMessage
Mapping	MappingNotify
Timer	Toolkit Timer Event
EventTimer	Toolkit ‘‘Event Timer’’ event

Supported Abbreviations:

Abbreviation	Meaning
Ctrl	KeyPressed with control modifier.
Meta	KeyPressed with meta modifier.
Shift	KeyPressed 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. In the event that no event bindings are specified externally, the widget class specifies the default bindings in textual form.

Appendix C: Intrinsic.h 25. Appendix C: Intrinsic.h

```
/*
* $Header: Intrinsic.h,v 1.32 87/09/13 20:36:19 newman Exp $
*/

/*
 * Copyright 1987 by Digital Equipment Corporation, Maynard, Massachusetts.
 *
 *                      All Rights Reserved
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose and without fee is hereby granted,
 * provided that the above copyright notice appear in all copies and that
 * both that copyright notice and this permission notice appear in
 * supporting documentation, and that the name of Digital Equipment
 * Corporation not be used in advertising or publicity pertaining to
 * distribution of the software without specific, written prior permission.
 *
 *
 * DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
 * ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL
 * DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
 * ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
 * WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
 * SOFTWARE.
 */
#ifndef _Xtintrinsic_h
#define _Xtintrinsic_h


/****************************************************************
 ****************************************************************
 ***                                                          ***
 ***                                                          ***
 ***                   X Toolkit Intrinsics                   ***
 ***                                                          ***
 ***                                                          ***
 ****************************************************************
 ****************************************************************/
/****************************************************************
 *
 * Miscellaneous definitions
 *
 ****************************************************************/


#include        <X11/Xlib.h>
#include        <X11/Xresource.h>
#include        <sys/types.h>

#ifndef NULL
#define NULL 0
#endif
```

```
#ifndef FALSE
#define FALSE 0
#define TRUE  1
#endif

#define XtNumber(arr)                           ((Cardinal) (sizeof(arr) / sizeof(arr[0])))
#define XtOffset(type,field)    ((unsigned int)&(((type)NULL)->field))
typedef char *String;
typedef struct _WidgetRec *Widget;
typedef struct _WidgetClassRec *WidgetClass;
typedef struct _CompositeRec *CompositeWidget;
typedef struct _XtEventRec *_XtEventTable;
typedef struct _XtActionsRec *XtActionList;
typedef struct _XtResource *XtResourceList;
typedef struct _GrabRec  *GrabList;
typedef unsigned int   Cardinal;
typedef char                            Boolean;
typedef unsigned long                   *Opaque;
typedef struct _TranslationData         *_XtTranslations;
typedef struct _XtCallbackRec*    XtCallbackList;
typedef unsigned long   XtCallbackKind;
typedef unsigned long   XtValueMask;
typedef unsigned long   XtIntervalId;
typedef unsigned int    XtGeometryMask;
typedef unsigned long   XtGCMask;   /* Mask of values that are used by widget*/
typedef unsigned long   Pixel;          /* Index into colormap        */
typedef int                             Position;   /* Offset from 0 coordinate        */
typedef unsigned int                    Dimension;  /* Size in pixels        */
                                          /* should be unsigned, but pcc       */
                                          /* generates bad code for unsigned? */

typedef void (*XtProc)();
   /* takes no arguments */

typedef void (*XtWidgetProc)();
   /* Widget widget */

typedef void (*XtArgsProc)();
   /* Widget widget */
   /* ArgList args */
   /* Cardinal num_args */

typedef void (*XtInitProc)();
   /* Widget requestWidget; */
   /* Widget newWidget; */
   /* ArgList args */
   /* Cardinal num_args */

typedef Boolean (*XtSetValuesProc)();  /* returns TRUE if redisplay needed */
   /* Widget widget;     */
   /* Widget request;                          */
   /* Widget new;        */
   /* Boolean last;      */
```

```
typedef void (*XtExposeProc)();
   /* Widget    widget; */
   /* XEvent    *event; */

typedef void (*XtRealizeProc) ();
   /* Widget                                    widget;    */
   /* XtValueMask mask;                           */
   /* XSetWindowAttributes *attributes;    */

typedef enum  {
   XtGeometryYes,       /* Request accepted. */
   XtGeometryNo,        /* Request denied. */
   XtGeometryAlmost,    /* Request denied, but willing to take replyBox. */
} XtGeometryResult;

typedef XtGeometryResult (*XtGeometryHandler)();
   /* Widget                                      widget */
   /* XtWidgetGeometry       *request */
   /* XtWidgetGeometry                            *reply   */

/****************************************************************
 *
 * System Dependent Definitions
 *
 *
 * XtArgVal ought to be a union of caddr_t, char *, long, int *, and proc *
 * but casting to union types is not really supported.
 *
 * So the typedef for XtArgVal should be chosen such that
 *
 *     sizeof (XtArgVal) >=                     sizeof(caddr_t)
 *                                              sizeof(char *)
 *                                              sizeof(long)
 *                                              sizeof(int *)
 *                                              sizeof(proc *)
 *
 * ArgLists rely heavily on the above typedef.
 *
 ****************************************************************/
typedef long XtArgVal;

/****************************************************************
 * Widget Core Data Structures
 *
 *
 ****************************************************************/

typedef struct _CorePart {
   WidgetClass                                  widget_class;/* pointer to Widget's ClassRec     */
   Widget                                       parent;/* parent widget       */
   String        name;                        /* widget resource name     */
   XrmName       xrm_name;                    /* widget resource name quarkified   */
   Screen                                       *screen;/* window's screen     */
   Window                                       window;/* window ID     */
```

```
   Position       x, y;                          /* window position     */
   Dimension      width, height;                 /* window dimensions     */
   Cardinal       depth;                         /* number of planes in window       */
   Dimension      border_width;                  /* window border width     */
   Pixel    border_pixel;                        /* window border pixel     */
   Pixmap         border_pixmap;                 /* window border pixmap or NULL      */
   Pixel    background_pixel;                    /* window background pixel     */
   Pixmap         background_pixmap;             /* window background pixmap or NULL  */
   _XtEventTable   event_table;                  /* private to event dispatcher       */
   _XtTranslations translations;                 /* private to Translation Manager    */
   Boolean        visible;                       /* is window mapped and not occluded?*/
   Boolean                                         sensitive;/* is widget sensitive to user events*/
   Boolean        ancestor_sensitive;            /* are all ancestors sensitive?      */
   Boolean        managed;                       /* is widget geometry managed?    */
   Boolean                                         mapped_when_managed;/* map window if it's managed?      */
   Boolean        being_destroyed;               /* marked for destroy     */
   XtCallbackList  destroy_callbacks;            /* who to call when widget destroyed */
} CorePart;

typedef struct _WidgetRec {
   CorePart    core;
 } WidgetRec;

typedef Widget *WidgetList;

/******************************************************************
 *
 * Core Class Structure. Widgets, regardless of their class, will have
 * these fields.  All widgets of a given class will have the same values
 * for these fields.  Widgets of a given class may also have additional
 * common fields.  These additional fields are included in incremental
 * class structures, such as CommandClass.
 *
 * The fields that are specific to this subclass, as opposed to fields that
 * are part of the superclass, are called "subclass fields" below.  Many
 * procedures are responsible only for the subclass fields, and not for
 * any superclass fields.
 *
 ********************************************************************/

typedef struct _CoreClassPart {
   WidgetClass                                     superclass;       /* pointer to superclass ClassRec     */
   String         class_name;                        /* widget resource class name         */
   Cardinal       widget_size;      /* size in bytes of widget record      */
   XtProc                                          class_initialize;  /* class initialization proc     */
   Boolean        class_inited;     /* has class been initialized?        */
   XtInitProc     initialize;                        /* initialize subclass fields         */
   XtRealizeProc   realize;                          /* XCreateWindow for widget     */
   XtActionList    actions;                          /* widget semantics name to proc map  */
   Cardinal                                        num_actions;       /* number of entries in actions       */
   XtResourceList  resources;                        /* resources for subclass fields      */
   Cardinal       num_resources;     /* number of entries in resources      */
   XrmClass       xrm_class;                         /* resource class quarkified     */
   Boolean        compress_motion;   /* compress MotionNotify for widget   */
```

```
   Boolean        compress_exposure; /* compress Expose events for widget  */
   Boolean        visible_interest;  /* select for VisibilityNotify        */
   XtWidgetProc   destroy;                    /* free data for subclass pointers    */
   XtWidgetProc   resize;                     /* geom manager changed widget size   */
   XtExposeProc   expose;                     /* rediplay window     */
   XtSetValuesProc set_values;                /* set subclass resource values      */
   XtWidgetProc   accept_focus;     /* assign input focus to widget       */
  } CoreClassPart;

typedef struct _WidgetClassRec {
   CoreClassPart core_class;
} WidgetClassRec;

extern WidgetClassRec widgetClassRec;
extern WidgetClass widgetClass;

/***************************************************************************
 *
 * Additional instance fields for widgets of (sub)class 'Composite'
 *
 ***************************************************************************/

typedef Cardinal (*XtOrderProc)();
   /* Widget child; */


typedef struct _CompositePart {
   WidgetList  children;                      /* array of ALL widget children     */
   Cardinal    num_children;                  /* total number of widget children     */
   Cardinal    num_slots;         /* number of slots in children array    */
   Cardinal    num_mapped_children; /* count of managed and mapped children */
   XtOrderProc insert_position;     /* compute position of new child     */
} CompositePart;

typedef struct _CompositeRec {
   CorePart      core;
   CompositePart composite;
} CompositeRec;

typedef struct _ConstraintPart {
   caddr_t     mumble;                        /* No new fields, keep C compiler happy */
} ConstraintPart;

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

/********************************************************************
 *
 *  Additional class fields for widgets of (sub)class 'Composite'
 *
 ********************************************************************/
```

```
typedef struct _CompositeClassPart {
   XtGeometryHandler geometry_manager;         /* geometry manager for children   */
   XtWidgetProc      change_managed;           /* change managed state of child   */
   XtArgsProc                                     insert_child;  /* physically add child to parent  */
   XtWidgetProc      delete_child;             /* physically remove child     */
   XtWidgetProc      move_focus_to_next; /* move Focus to next child     */
   XtWidgetProc      move_focus_to_prev; /* move Focus to previous child    */
} CompositeClassPart;

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

extern CompositeClassRec compositeClassRec;
extern CompositeWidgetClass compositeWidgetClass;


typedef struct _ConstraintClassPart {
   XtResourceList constraints;                   /* constraint resource list     */
   Cardinal   num_constraints;       /* number of constraints in list       */
} ConstraintClassPart;

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

extern ConstraintClassRec constraintClassRec;
extern ConstraintWidgetClass constraintWidgetClass;

/****************************************************************************
 *
 * Generic Procedures
 *
 ****************************************************************************/


extern Boolean XtIsSubclass ();
   /* Widget       widget;                          */
   /* WidgetClass  widgetClass;    */

/* Some macros to get frequently used components of a widget */

#define XtDisplay(widget)              ((widget)->core.screen->display)
#define XtScreen(widget)               ((widget)->core.screen)
#define XtWindow(widget)               ((widget)->core.window)
#define XtMapWidget(widget)            XMapWindow(XtDisplay(widget), XtWindow(widget))
#define XtUnmapWidget(widget)          XUnmapWindow(XtDisplay(widget), XtWindow(widget))
#define XtIsComposite(widget)          XtIsSubclass(widget, (WidgetClass)compositeWidgetClass)
#define XtClass(widget)                ((widget)->core.widget_class)
#define XtSuperclass(widget)           (XtClass(widget)->core_class.superclass)
```

```
extern Widget XtCreateWidget ();
   /* String                                    name;    */
   /* WidgetClass  widgetClass;    */
   /* Widget      parent;                       */
   /* ArgList     args;                         */
   /* Cardinal    num_args;      */

extern Widget TopLevelCreate (); /*hack for now*/
   /* String                                    name; */
   /* WidgetClass widgetClass; */
   /* Screen      *screen;*/
   /* ArgList     args; */
   /* Cardinal    num_args; */



extern void XtRealizeWidget ();
   /* Widget    widget     */

extern Boolean XtIsRealized ();
   /* Widget    widget; */

extern void XtDestroyWidget ();
   /* Widget widget */

extern void XtSetSensitive ();
   /* Widget    widget;    */
   /* Boolean   sensitive; */

extern void XtSetMappedWhenManaged ();
   /* Widget    widget;    */
   /* Boolean   mappedWhenManaged; */

/**********************************************************
 *
 * Composite widget Procedures
 *
 **********************************************************

extern void XtManageChildren ();
   /* WidgetList children; */
   /* Cardinal   num_children; */

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

extern void XtUnmanageChildren ();
   /* WidgetList children; */
   /* Cardinal   num_children; */

extern void XtUnmanageChild ();
   /* Widget child; */
```

```
/************************************************************
 *
 *  Callbacks
 *
 ************************************************************/

typedef void (*XtCallbackProc)();
   /* Widget widget; */
   /* caddr_t closure;  data the application registered */
   /* caddr_t callData; widget instance specific data passed to application*/

typedef struct _XtCallbackRec {
   XtCallbackList next;
   Widget   widget;
   XtCallbackProc callback;
   Opaque  closure;
}XtCallbackRec;

extern XtCallbackKind XtNewCallbackKind();
   /* WidegtClass widgetClass; */
   /* Cardinal offset; */

extern void XtAddCallback ();
   /* Widget      widget;   */
   /* XtCallbackKind callbackKind; */
   /* XtCallbackProc callback; */
   /* caddr_t      closure;  */

extern void XtRemoveCallback ();
   /* Widget      widget;   */
   /* XtCallbackKind callbackKind; */
   /* XtCallbackProc callback; */
   /* caddr_t      closure;  */


extern void XtRemoveAllCallbacks ();
   /* Widget widget; */
   /* XtCallbackKind callbackKind; */

extern void XtCallCallbacks ();
   /* Widget  widget; */
   /* XtCallbackKind callbackKind; */
   /* caddr_t callData */

/***************************************************************
 *
 * Toolkit initialization
 *
 ***************************************************************/

extern Widget XtInitialize();
   /* XtAtom                                  name;      */
   /* XtAtom                                  class;     */
   /* XrmOptionsDescRec    options;    */
```

```
   /* Cardinal           num_options;  */
   /* Cardinal                                        *argc; */ /* returns count of args not processed */
   /* char                                            **argv;     */

/****************************************************************
 *
 * Memory Management
 *
 ****************************************************************/

extern char *XtMalloc(); /* size */
   /* Cardinal size; */

extern char *XtCalloc(); /* num, size */
   /* Cardinal num, size; */

extern char *XtRealloc(); /* ptr, num */
   /* char     *ptr; */
   /* Cardinal num; */

extern void XtFree(); /* ptr */
         /* char  *ptr */


/****************************************************************
 *
 * Arg lists
 *
 ****************************************************************/

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

#define XtSetArg(arg, n, d)     ( (arg).name = (n), (arg).value = (XtArgVal)(d) )

extern ArgList XtMergeArgLists(); /* args1, num_args1, args2, num_args2 */
   /* ArgList args1;       */
   /* int     num_args1;   */
   /* ArgList args2;       */
   /* int     num_args2;   */



/****************************************************************
 *
 * Event Management
 *
 ****************************************************************/

/* ||| Much of this should be private */
XtCallbackList DestroyList;
Display *toplevelDisplay;
```

```
typedef unsigned long EventMask;

typedef enum {pass,ignore,remap} GrabType;
typedef void (*XtEventHandler)(); /* widget, closure, event */
   /* Widget  widget   */
   /* caddr_t closure  */
   /* XEvent  *event;  */

typedef struct _XtEventRec {
    _XtEventTable next;
    EventMask   mask;
    Boolean     non_filter;
    XtEventHandler proc;
    Opaque closure;
}XtEventRec;

typedef struct _GrabRec {
   GrabList next;
   Widget  widget;
   Boolean  exclusive;
}GrabRec;

typedef struct _MaskRec {
   EventMask   mask;
   GrabType    grabType;
   Boolean     sensitive;
}MaskRec;
#define is_sensitive TRUE
#define not_sensitive FALSE
GrabRec *grabList;

extern EventMask _XtBuildEventMask(); /* widget */
   /* Widget widget; */

extern void XtAddEventHandler(); /* widget, eventMask, other, proc, closure */
   /* Widget                                   widget      */
   /* EventMask       eventMask;  */
   /* Boolean         other;      */
   /* XtEventHandler   proc;       */
   /* caddr_t                                  closure ;   */


extern void XtRemoveEventHandler(); /* widget,eventMask,other,proc,closure */
   /* Widget                                   widget      */
   /* EventMask       eventMask;  */
   /* Boolean         other;      */
   /* XtEventHandler   proc;       */
   /* caddr_t                                  closure ;   */


extern void XtDispatchEvent(); /* event */
   /* XEvent                                   *event; */

extern void XtMainLoop();
```

```
/******************************************************************
 *
 * Event Gathering Routines
 *
 ******************************************************************/

typedef unsigned long                   XtInputMask;

#define XtInputNoneMask                 0L
#define XtInputReadMask                 (1L<<0)
#define XtInputWriteMask                (1L<<1)
#define XtInputExceptMask               (1L<<2)

extern Atom XtHasInput;
extern Atom XtTimerExpired;

extern XtIntervalId XtAddTimeOut();
   /* Widget widget;                    */
   /* unsigned long interval;  */

extern void XtRemoveTimeOut();
   /* XtIntervalId timer;      */

extern unsigned long XtGetTimeOut();
   /* XtIntervalId   timer;    */

extern void XtAddInput(); /* widget, source, condition */
   /* Widget widget                     */
   /* int source;                       */
   /* XtInputMask inputMask;            */

extern void XtRemoveInput(); /* widget, source, condition */
   /* Widget widget                     */
   /* int source;                       */
   /* XtInputMask inputMask;            */

extern void XtNextEvent(); /* event */
   /* XtEvent *event;                   */

extern XtPeekEvent(); /* event */
   /* XtEvent *event;                   */

extern Boolean XtPending ();


/******************************************************************
 *
 * Geometry Management
 *
 ******************************************************************/

#define XtDontChange                    5 /* don't change the stacking order stack_mode */

typedef struct {
```

```
    XtGeometryMask request_mode;
    Position x, y;
    Dimension width, height, border_width;
    Widget sibling;
    int stack_mode;                         /* Above, Below, TopIf, BottomIf, Opposite */
} XtWidgetGeometry;


extern XtGeometryResult XtMakeGeometryRequest();
    /*  widget, request, reply                      */
    /* Widget                               widget;     */
    /* XtWidgetGeometry    *request;                */
    /* XtWidgetGeometry                     *reply;  /* RETURN */

extern XtGeometryResult XtMakeResizeRequest ();
    /* Widget    widget;                    */
    /* Dimension width, height; */
    /* Dimension *replyWidth, *replyHeight; */

extern void XtResizeWindow(); /* widget */
    /* Widget widget; */

extern void XtResizeWidget(); /* widget, width, height, borderWidth */
    /* Widget  widget */
    /* Dimension width, height, borderWidth; */

extern void XtMoveWidget(); /* widget, x, y */
    /* Widget  widget */
    /* Position x, y  */


/****************************************************************
 *
 * Graphic Context Management
 *****************************************************************/

extern GC XtGetGC(); /* widget, valueMask, values */
    /* Widget    widget */
    /* XtGCMask valueMask; */
    /* XGCValues *values; */

extern void XtDestroyGC ();
    /* GC gc; */

/****************************************************************
 *
 * Resources
 *
 ****************************************************************/

#define StringToQuark(string) XrmAtomToQuark(string)
#define StringToName(string) XrmAtomToName(string)
#define StringToClass(string) XrmAtomToClass(string)
```

```
typedef struct _XtResource {
   String    resource_name;                /* Resource name    */
   String    resource_class;               /* Resource class    */
   String    resource_type;                /* Representation type desired    */
   Cardinal   resource_size;               /* Size in bytes of representation    */
   Cardinal   resource_offset;/* Offset from base to put resource value   */
   String    default_type;                 /* representation type of specified default */
   caddr_t    default_addr;   /* Address of default resource    */
} XtResource;


extern void XtGetResources();
   /* Widget      widget;          */
   /* ArgList                                args;*/
   /* int    num_args;                     */

extern void XtReadBinaryDatabase ();
   /* FILE    *f;                          */
   /* ResourceDatabase *db;                */

extern void XtWriteBinaryDatabase ();
   /* FILE    *f;                          */
   /* ResourceDatabase db;                 */

extern void XtSetValues();
   /* Widget         widget;       */
   /* ArgList                              args;*/
   /* int                                  num_args;      */

extern void XtGetValues();
   /* Widget         widget;       */
   /* ArgList                              args;*/
   /* Cardinal                             num_args;      */

extern Widget XtStringToWidget ();
   /* String s; */

extern WidgetClass XtStringToClass ();
   /* String s; */



/****************************************************************
 *
 * Translation Management
 *
 ****************************************************************/

typedef struct _XtActionsRec{
   char    *string;
   caddr_t value;
} XtActionsRec;

/* ||| Should be private */
```

```
extern void XtDefineTranslation ();
   /* Widget widget */

/**************************************************************
 *
 * Error Handling
 *
 *************************************************************/


extern void XtSetErrorHandler(); /* errorProc */
  /* (*errorProc)(String); */

extern void XtError();  /* message */
   /* String message */

extern void XtSetWarningHandler(); /* errorProc */
  /* (*errorProc)(String); */

extern void XtWarning();  /* message */
   /* String message */


#endif _Xtintrinsic_h
/* DON'T ADD STUFF AFTER THIS #endif */
```