Elk/Xt Reference Manual

Oliver Laumann

1. Introduction

This manual describes the functions, special forms, and variables defined by the Xt (X Toolkit Intrinsics) extension included in the Elk distribution. Most of the functions are directly equivalent to a function of the X toolkit C library, so that the description need not be repeated. In such cases, only the name of the corresponding Xt function is mentioned. Thus, you should have the *X Toolkit Intrinsics − C Language Interface* manual within reach when using this reference manual.

The functions listed in this document are loaded when the expression

```
(require 'xwidgets)
```

is evaluated or, when the OSF/Motif software has been installed on your system and you want to use Motif widgets from within Scheme, when

```
(require 'motif)
```

is evaluated in the interpreter's top level or in a Scheme program. If you only want to use the toolkit functionality (and no widgets), evaluate

```
(require 'xt).
```

Note that all of the above forms cause the Elk/Xlib functions to be loaded as well.

Individual widgets are loaded by evaluating

(load-widgets . *widget-names*)

Each *widget-name* is a symbol (not quoted, since *load-widgets* is a macro).

The widgets are loaded from subdirectories of "$install_dir/runtime/obj" (where $install_dir is the directory where you have installed Elk on your system).

In the following, the types of arguments of the listed procedures are not specified when they are obvious from the context or from the name. For instance, an argument named *widget* is always of type *widget*, an argument named *context* is an object of type *context* (application context), etc. Arguments the names of which end in "?" are always of type *boolean*.

In the following, each description of a procedure, special form, or variable lists the kind of object in boldface. Here, **procedure** denotes either a primitive procedure or a compound procedure, **syntax** denotes a special form or a macro, and **variable** denotes a global variable that has some initial value and can be re-assigned a new value by the user (by means of *set!* or *fluid-let*).

2. Widget Classes

(class? *x***)** **procedure**

Returns #t iff *x* is an object of type *class* (widget class).

(find-class *name-of-class***)** **procedure**

Returns the widget class of the specified name (an object of type *class*). *name-of-class* is a string or a symbol.

(class-resources *widget-class***)** **procedure**

See *XtGetResourceList*. Returns the resource list of the specified widget class. Each element of the list is a list of three symbols – the resource name, the resource class, and the resource type.

(class-constraint-resources *widget-class***)** **procedure**

See *XtGetConstraintRespourceList*. Returns the list of constraint resources that are defined for the specified widget class. Each element of the list is a list of three symbols – the resource name, the resource class, and the resource type.

(class-sub-resources *widget-class***)** **procedure**

Returns the list of sub-resources (if there are any) of the specified widget class. See *class-resources* above.

(class-exists? *name-of-class***)** **procedure**

Returns #t iff a widget class of the given name exists (i. e. has been loaded by means of *load-widgets*). *name-of-class* is a string or a symbol.

3. Widget Functions

(widget? *x***)** **procedure**

Returns #t iff *x* is an object of type *widget*.

(destroy-widget *widget***)** **procedure**

See *XtDestroyWidget*.

(create-shell *application-name application-class parent display . args***)** **procedure**

See *XtAppCreateShell*. *application-name* and *application-class* are strings or symbols or #f (NULL is used in the latter case). *parent* is a widget. The number of *args* must be even, the 1st, 3rd, etc. argument is the name of a resource to be set (a string or a symbol), the 2nd, 4th, etc. argument is the corresponding value.

(create-widget *widget-class parent . args***)** **procedure**
(create-widget *widget-name widget-class parent . args***)** **procedure**

See *XtCreateWidget*. *widget-name* is a string or a symbol; *parent* is a widget. If no *widget-name* is given, the name of the widget class is used. The number of *args* must be even, the 1st, 3rd, etc. argument is the name of a resource to be set (a string or a symbol), the 2nd, 4th, etc. argument is the corresponding value.

(create-managed-widget . *args***)** **procedure**

Applies *create-widget* to the arguments and then calls *manage-child* with the newly created widget.

(realize-widget *widget***)** **procedure**

See *XtRealizeWidget*.

(unrealize-widget *widget***)** **procedure**

See *XtUnrealizeWidget*.

(widget-realized? *widget***)** **procedure**

See *XtIsRealized*.

(widget-display *widget***)** **procedure**

See *XtDisplay*. Returns an object of type *display*.

(widget-parent *widget***)** **procedure**

See *XtParent*.

(widget-name *widget***)** **procedure**

See *XtName*. Returns the name of a widget as a string.

(widget−>window *widget***)** **procedure**
(widget-window *widget***)** **procedure**

See *XtWindow*. Returns an object of type *window*.

(widget-composite? *widget***)** **procedure**

See *XtIsComposite*.

(manage-children . *widgets***)** **procedure**

See *XtManageChildren*.

(manage-child *widget***)** **procedure**

Calls *manage-children* with the specified widget.

(unmanage-children . *widgets***)** **procedure**

See *XtUnmanageChildren*.

(unmanage-child *widget***)** **procedure**

Calls *unmanage-children* with the specified widget.

(widget-managed? *widget***)** **procedure**

See *XtIsManaged*.

(widget-class *widget***)** **procedure**

See *XtClass*. Returns an object of type *class*.

(widget-superclass *widget***)** **procedure**

See *XtSuperclass*. Returns an object of type *class* or the symbol `none` when the widget's class does not have a super-class.

(widget-subclass? *widget class***)** **procedure**

See *XtIsSubclass*.

(set-mapped-when-managed! *widget managed?***)** **procedure**

See *XtSetMappedWhenManaged*.

(map-widget *widget***)** **procedure**

See *XtMapWidget*.

(unmap-widget *widget***)** **procedure**

See *XtUnmapWidget*.

(set-values! *widget . args***)** **procedure**

See *XtSetValues*. The number of *args* must be even, the 1st, 3rd, etc. argument is the name of a resource to be set (a string or a symbol), the 2nd, 4th, etc. argument is the corresponding value.

(get-values *widget . args***)** **procedure**

See *XtGetValues*. Each *arg* is the name of a resource (a string or a symbol). Returns a list of the values of the specified resources.

(widget-context *widget***)** **procedure**

See *XtWidgetToApplicationContext*. Returns an object of type *context*.

(set-sensitive! *widget sensitive?***)** **procedure**

See *XtSetSensitive*.

(widget-sensitive? *widget***)** **procedure**

See *XtIsSensitive*.

(window−>widget *window***)** **procedure**

See *XtWindowToWidget*.

(name−>widget *root-widget name***)** **procedure**

See *XtNameToWidget*. *name* is a string or a symbol.

(widget-translate-coordinates *widget x y***)** **procedure**

See *XtTranslateCoords*. Returns the root-relative x and y coordinates as a pair of integers.

4. Callback Functions

(add-callbacks *widget callback-name . callback-functions***)** **procedure**

See *XtAddCallbacks*. Adds the functions to a callback list of the specified widget. *callback-name* is a string or a symbol. Each callback function will be called with at least one argument − the widget to which the function has been attached.

(add-callback *widget callback-name callback-function***)** **procedure**

Calls *add-callbacks* with the specified function.

5. Popup Shells

(create-popup-shell *widget-class parent-widget . args***)** **procedure**
(create-popup-shell *widget-name widget-class parent-widget . args***)** **procedure**

See *XtCreatePopupShell*. *widget-name* is a string or a symbol. If no widget name is given, the name of the shell class is used. The number of *args* must be even, the 1st, 3rd, etc. argument is the name of a resource to be set (a string or a symbol), the 2nd, 4th, etc. argument is the corresponding value.

(popup *shell-widget grab-kind***)** **procedure**

See *XtPopup*. *grab-kind* is a symbol (`grab-once`, `grab-nonexclusive`, or `grab-exclusive`).

(popdown *shell-widget***)** **procedure**

See *XtPopdown*.

6. Application Contexts

(context? *x***)** **procedure**

Returns #t iff *x* is an object of type *context* (application context).

(create-context) **procedure**

See *XtCreateApplicationContext*.

(destroy-context *context***)** **procedure**

See *XtDestroyApplicationContext*.

(initialize-display *context display application-name application-class***)** **procedure**

See *XtDisplayInitialize*, *XtOpenDisplay*. If *display* is an object of type *display*, *XtDisplayInitialize* is called. If *display* is a display name (a string or a symbol) or #f, *XtOpenDisplay* is called (with a NULL display in the latter case), and the newly opened display is returned. *application-name* and *application-class* are strings or symbols or #f (NULL and the empty string are used in the latter case).

(application-initialize *name . fallback-resources***)** **procedure**

A convenience function that creates an application context by a call to *create-context*, sets the fallback resources (if any), initializes the display by a call to *initialize-display* with the specified *name* and a class of #f, and creates and returns an application shell with the name *name* and class #f.

Calling *application-initialize* more than once may cause the application contexts and displays that were created by earlier calls to be closed during a future garbage collection.

(display–>context *display***)** **procedure**

See *XtDisplayToApplicationContext*.

(set-context-fallback-resources! *context . resources***)** **procedure**

See *XtAppSetFallbackResources*. Each *resource* is a string.

(context-main-loop *context***)** **procedure**

See *XtAppMainLoop*.

(context-pending *context***)** **procedure**

See *XtAppPending*. Returns a list of symbols (`x-event`, `timer`, `alternate-input`).

(context-process-event *context . mask***)** **procedure**

See *XtProcessEvent*. The optional argument *mask* is a list of symbols (see *context-pending* above). *XtIMAll* is used if the *mask* argument is omitted

(context-add-work-proc *context procedure***)** **procedure**

See *XtAppAddWorkProc*. Returns an *identifier* that can be used as an argument to *remove-work-proc*. *procedure* is a procedure with no arguments.

(remove-work-proc *identifier***)** **procedure**

See *XtRemoveWorkProc*. *identifier* must be the return value of a previous call to *context-add-work-proc*. Each such *identifier* can be used as an argument for *remove-work-proc* only once.

(context-add-timeout *context timer-value procedure***)** **procedure**

See *XtAppAddTimeOut*. *timer-value* is an integer. Returns an *identifier* that can be used as an argument to *remove-timeout*. The time-out procedure will be called with one argument, the identifier returned by the call to *context-add-timeout* (i. e. the object that uniquely identifies the timer).

(remove-timeout *identifier***)** **procedure**

See *XtRemoveTimeOut*. *identifier* must be the return value of a previous call to *context-add-timeout*. Each such *identifier* can be used as an argument for *remove-timeout* only once.

(context-add-input *context source procedure . mask***)** **procedure**

See *XtAppAddInput*. *source* is a file port. Returns an *identifier* that can be used as an argument to *context-remove-input*. The *procedure* will be called with two arguments − *source* and the identifier returned by the call to *context-add-input*.

The optional *mask* argument is a list of one or more of the symbols *read*, *write*, and *exception*. It specifies the condition on which the procedure will be called. If no *mask* argument is given, *read* is used if *source* is an input-port, *write* if it is an output-port, and both *read* and *write* if it is an input-output-port.

(remove-input *identifier***)** **procedure**

See *XtRemoveInput*. *identifier* must be the return value of a previous call to *context-add-input*. Each such *identifier* can be used as an argument for *remove-input* only once.

(identifier? *x***)** **procedure**

Returns #t iff *x* is an *identifier* (an object returned by functions like *context-add-timeout*).

7. Translations Management Functions

(context-add-action *context name procedure***)** **procedure**

See *XtAppAddActions*. *name* is the name of the action (a string or a symbol). The action procedure will be invoked with three arguments: a widget, a list of event-specific arguments (see *handle-events*) and a list of strings (the action arguments).

(augment-translations *widget translation-table***)** **procedure**

See *XtAugmentTranslations*. *translation-table* is a string; *XtParseTranslationTable* is applied to it.

(override-translations *widget translation-table***)** **procedure**

See *XtOverrideTranslations*. *translation-table* is a string; *XtParseTranslationTable* is applied to it.

(uninstall-translations *widget***)** **procedure**

See *XtUninstallTranslations*.

(install-accelerators *dst-widget src-widget***)** **procedure**

See *XtInstallAccelerators*.

(install-all-accelerators *dst-widget src-widget***)** **procedure**

See *XtInstallAllAccelerators*.

(multi-click-time *display***)** **procedure**

See *XtGetMultiClickTime*. Returns an integer.

(set-multi-click-time! *display time***)** **procedure**

See *XtSetMultiClickTime*. *time* is an integer.

8. Error Handling

xt-warning-handler **variable**

See *XtSetWarningHandler*. When a warning message is to be printed by the Xt intrinsics and the global variable *xt-warning-handler* is bound to a compound procedure, this procedure is invoked with the error message (a string) as an argument. When this variable is not bound to a compound procedure, the message is sent to the current output port. The initial value of this variable is the empty list.

This interface is bogus and will be replaced by a more useful mechanism in future versions of the software.

9. Miscellaneous Functions

(xt-release-4-or-later?) **procedure**

Returns always #t.

(xt-release-5-or-later?) **procedure**

Returns #t iff the Xt extension is linked together with the X11 Release 5 toolkit intrinsics or later versions of the intrinsics.

(xt-release-6-or-later?) **procedure**

Returns #t iff the Xt extension is linked together with the X11 Release 6 toolkit intrinsics or later versions of the intrinsics.

10. Interaction with the Garbage Collector

The Scheme garbage collector destroys objects of type *context* or *widget* that are not longer accessible from within the Scheme program. This is done by invoking the function *destroy-context* or *destroy-widget*, respectively, with the unreferenced object as an argument.

The garbage collector only destroys objects that have been created from with the Scheme program (by functions like *create-context* or *create-widget*). Objects that have been obtained from Xt through functions like *widget-context* (and are owned by the Xt internals), are ignored by the garbage collector.

Programmers must make sure that an object is accessible during the object's entire lifetime, otherwise future runs of the garbage collector can result in undesired termination of the object. One must be especially careful when results of functions that create new objects (such as *create-context*) are ignored or assigned to local variables as in

```
(define (initialize)
  (let* ((con (create-context))
         (dpy (initialize-display con #f 'Test #f)))
    (create-shell 'Test #f (find-class 'application-shell) dpy))))
```

In this example, after termination of the function, the garbage collector will destroy the application context created by the call to *create-context* as well as the display, as they are no longer directly accessible from within the program. Bugs like this are often hard to find, since (in the above example) the shell widget returned by *initialize* can still be used, although its application context and the display associated with the application context have already been destroyed.

The problem can be solved by re-writing the above function like this:

```
(define initialize #f)

(let ((con) (dpy))
  (set! initialize
    (lambda ()
      (set! con (create-context))
      (set! dpy (initialize-display con #f 'Test #f))
      (create-shell 'Test #f (find-class 'application-shell) dpy))))
```

An alternative solution is to place the application context and display into a global variable, so that they can be terminated explicitly by the program when desired.

Index

Table of Contents