24
Error Recovery

This chapter addresses the general-usage Error Handling facilities that apply to all XView packages. All XView packages, including extensions to existing packages or those packages you write yourself, should address error handling to provide adequate feedback for the programmer.

There are two different types of errors addressed: XView errors and Xlib protocol errors. The two are very different from one another; XView errors are generated by misusing the XView Toolkit in some way, while Xlib protocol errors result from calling Xlib functions incorrectly or using incorrect values to those functions. We'll start with XView errors since, as you may already know, it is easy to generate run-time errors when writing XView applications.

24.1 XView Errors

XView errors are caused by specifying invalid or unknown attributes, objects, or values. Often, XView errors are generated by simply failing to terminate a list of values of attribute-value pairs. Whatever the reason, for an error, the XView internals call the function `xv_error()`.

This function is the highest level interface into the XView error package. Like many of the other XView functions, `xv_error()` takes a `NULL`-terminated attribute-value list of parameters. These parameters, set by the calling function, describe the nature of the error encountered. `xv_error()` then calls either a programmer-supplied function or a default function that prints the nature of the error to `stderr`. Before discussing the details of these routines, let's first address the problem of when you need to use them and to what extent.

XView error handling is most useful during the early development of applications. If you want to force a core dump for debugging purposes or if you want to suppress the standard error messages from being displayed on the controlling tty's `stderr`, then all you need to do is install a general error handling routine. If you are writing your own XView packages or wish to get detailed information about the nature of an XView error, see Section 24.4, "Advanced Error Handling."

First, we'll address the specific task of how to install an error handler for an XView application, then we'll move on to more advanced error handling methods.

24.2 Simple Error Handling

An error handler is installed using the attribute, XV_ERROR_PROC in the call to xv_init().

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
    XV_ERROR_PROC,   my_error_proc,
    NULL);
```

Because xv_init() is called only once, the error routine cannot be changed or uninstalled. The routine must be installed at this time, and it remains installed for the life of the application.

The XView error handler is called whenever xv_error() is called. That is, when there is an internal error in XView, not if there is an X Protocol error or errors of any other kind.

The following call generates an XView error:

```
xv_set(NULL, PANEL_LABEL, "foo", NULL);
```

The reason for the error is that an invalid XView object was given to xv_set(). This may seem like a simplistic example because no one would ever intentionally pass NULL as an object. However, attempting to set attributes for uninitialized objects is not an uncommon error. A more common mistake made by the novice XView programmer is forgetting to use a NULL terminator at the end of the attribute-value list to xv_set() or xv_create(). This is also a more subtle error that may not always generate an error due to the undefined value of the missing parameter.*

When xv_error() is called, it in turn calls the registered error handling routine, which takes the following form:

```
error_proc(object, avlist)
    Xv_object       object;
    Attr_attribute  avlist[ATTR_STANDARD_SIZE];
```

Since XView errors are always generated in response to an internal XView package or routine, there is always an object associated with the error. This is passed as the object parameter. The avlist provides details about the nature of the error. Note that this is *not* an attribute-value list in the form of the other attribute-value lists used throughout most of the book. It is an array of attributes and values that have been constructed from a NULL-terminated attribute-value list. See Chapter 25, *XView Internals*, for more details.

*A fundamental understanding of variable-argument lists shows that a missing parameter (the NULL parameter in this case) translates into an unpredictable value, which by many coincidences may turn out to be 0.

The following code fragment shows a sample error procedure that supports this interface.

```
error_proc(object, avlist)
Xv_object         object;
Attr_avlist       avlist;
{
    char buf[32];

    printf("%s\nDump core? ", xv_error_format(object, avlist));
    fflush(stdout);
    if (gets(buf) && (buf[0] == 'y' || buf[0] == 'Y'))
        abort(); /* may return if application is trapping SIGIOT */
    return XV_OK;
}
```

24.2.0.1 Using xv_error_format()

The above procedure makes use of the routine `xv_error_format()`. This function returns a pointer to a static `char *` describing the XView error that has occurred. It takes as parameters an XView object and an `Attr_avlist`. Because `xv_error_format()` returns a pointer to a static string, it should be copied into your own buffer if you wish to retain the value since repeated calls overwrite the contents.

Since the parameters to `xv_error_format()` are the same as those passed to the error function, they may be passed on to the format function without further processing. This function is useful if you don't want to parse the `Attr_avlist` yourself, but still wish to print the error message.

This is all very simple if you do not care to put a great deal of work into your error handling routine and parse the `avlist`. Note that we aren't even testing to see if the severity of the error was recoverable or not. If an error occurs, no matter how innocent, we want to trace it down to the offending function call.

24.3 X Error Handling

Catching errors that occur from Xlib or the X server should be done using the methods described in Volume One, *Xlib Programmer's Manual*. This section shows how you can write your own routine that handles Xlib errors as well as errors with the server.

To register an Xlib error handler, you set an XV_X_ERROR_PROC or you may use `XSetErrorHandler()`. Using `XSetErrorHandler()` allows you to register one error handling routine per application. `XSetErrorHandler()` overrides any existing error procedures and makes it necessary to deal with all X errors (by default XView catches some X error events). Using the attribute XV_X_ERROR_PROC in the `xv_init()` call allows XView to continue to filter out some X error events (some of these error events may be not fatal errors caused by the toolkit).

```
int x_error_proc();

xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv,
```

```
        XV_X_ERROR_PROC,  x_error_proc,
        NULL);
```

If an Xlib error occurs, your routine is called with the following parameters:

```
int
x_error_proc(dpy, event)
    Display      *dpy;
    XErrorEvent  *event;
```

The routine should return `XV_OK` if you have handled the error to your satisfaction. In this case, XView will ignore the error and continue on with the execution of the program. If the routine returns `XV_ERROR`, then XView calls its default error handler, prints an error message to `stderr`, and exits with a status of 1.

24.4 Advanced Error Handling

If you wish to examine in detail an XView error within your error handling procedure or are implementing your own XView package, the following sections may be useful. Each section addresses the error attributes and their associated values. Using the information described here along with Section 25.2, "Internal Attribute-Value Lists," in Chapter 25, *XView Internals*, you can construct a call to `xv_error()` and parse an error list passed to your error handling function.

24.4.0.1 Error types

The header file <*xview/xv_error.h*> contains the definitions of the types and functions used by the error package. These error types are used as parameters to `xv_error()` and from there to your error handling routine. The following list of attributes should be used if you are going to be calling `xv_error()` or if you wish to parse the `Attr_avlist` from the error handling routine.

`ERROR_BAD_ATTR`
: An attribute was specified that is not defined by XView. If the calling function forgets to terminate a list with a `NULL`, this is most likely the error value passed.

`ERROR_BAD_VALUE`
: A bad value was provided for an attribute. This includes out of range values, and so on. If you think the value given is correct, check its type. Passing `floats` when in fact they are read as `doubles` may cause this problem.

`ERROR_CANNOT_GET`
: `xv_get()` was used on an ungettable attribute.

`ERROR_CANNOT_SET`
: `xv_set()` was used on an unsettable attribute.

`ERROR_CREATE_ONLY`
: `xv_set()` was used to set an attribute that is only valid using `xv_create()`.

ERROR_INVALID_OBJECT
: The `object` parameter to the routine is invalid. Either the object was uninitialized or the object had been (or is in the process of being) destroyed.

ERROR_LAYER
: The layer of software that detected the error. Possible error layers are the operating system, the X server, the XView Toolkit, and the application.

ERROR_PKG
: The toolkit package that detected the error.

ERROR_SERVER_ERROR
: The error detected by the server; takes an `XErrorEvent *` as a value.

ERROR_SEVERITY
: The severity of the error detected. Its value is of type `Error_severity`. This is an enumerated type whose values may be ERROR_RECOVERABLE or ERROR_NON_RECOVERABLE. Unrecoverable errors should definitely cause the program to exit, whereas recoverable errors can cause an exit if you so desire.

ERROR_STRING
: Used by the calling function to `xv_error()` to give a description of the error if necessary. Trailing newlines are stripped.

24.4.1 Calling xv_error()

If you are trying to write your own XView package or add an extension to an existing package, you may need to call `xv_error()`. Calling it is similar to calling `xv_set()`; the first parameter is a handle to an XView object followed by a NULL-terminated attribute-value list consisting of the above attributes. It takes the form:

```
int
xv_error(object, attrs)
    Xv_object object;
```

`object` is the object for which the offending call had taken place. If the programmer called `xv_set()` on a `frame` object and the attributes passed were invalid, then you would call `xv_error()`, passing the `frame` as the object and a set of ERROR_ attribute-value pairs from the above list.

For example, the following call assumes the calling function tried to set an invalid attribute in the FRAME package. (Note that this is only an example, since the FRAME package does not do this.)

```
switch (attribute) {
    case FRAME_FOREGROUND :
        /* ... */
        break;
    case FRAME_OLD_RECT :
        xv_error(frame,
            ERROR_STRING,     "You cannot set this attribute.",
            ERROR_CANNOT_SET, attribute,
            ERROR_PKG,        FRAME,
```

```
                NULL);
            break;
        /* ... */
    }
```

As you can see, unless you are implementing the internals of an XView package, `xv_error()` may be of limited use. However, this example demonstrates how `xv_error()` can be called internally by XView.

24.4.1.1 Error severity

In the above example, the reason to call `xv_error()` is not a serious one, at least not one that should terminate the program. By default, calling `xv_error()` is a recoverable error, so in order to specify a non-recoverable error, the call to `xv_error()` should pass the attribute, ERROR_SEVERITY, and the value ERROR_NON_RECOVERABLE. Unrecoverable errors generally terminate using `exit()` with a non-zero exit status. However, `abort()` may be used instead to generate a core image used for debugging.

In any event, if the error handler is called, it should print a warning message and either continue or exit accordingly. If you install your own error routine, the choice is yours. Example 24-1 in the next section, shows how the error severity can be evaluated and acted upon accordingly.

24.4.2 Revisiting the Error Handler

Advanced usage of the ERROR package provides us with the ability to scan the attribute list in search of the causes of the error. With this information, you can print out more useful error messages or display them in a manner other than printing to `stderr`.

Recall that there are two parameters passed to the function:

```
error_proc(object, avlist)
    Xv_object       object;
    Attr_avlist     avlist;
```

`object` is the object in which the `xv_*` call failed to operate. You can get the type of `object` by calling:

```
Xv_pkg *pkg = (Xv_pkg *)xv_get(object, XV_TYPE);
```

If the error itself pertains to the object, then getting the type of the object may generate another error. You should not attempt to get the package until you test the error code in the `Attr_avlist` to be sure the error is not with the `object` parameter. If the error is not due to the object itself, the package returned by `xv_get()` indicates to which XView package the object belongs. This value matches the same argument as the second parameter to `xv_create()`. For example, the package returned may be MENU, CANVAS, PANEL_BUTTON, SERVER, etc.

The `Attr_avlist` may be scanned for the ERROR attributes. Example 24-1 shows how this is done. This is a very simplistic example and is for demonstration purposes only. A more complete example may be found in the function `xv_error_format()` in the XView source code.

Example 24-1. Example error parsing function

```
int
my_error_handler(object, avlist)
Xv_object    object;
Attr_avlist  avlist;
{
    Attr_avlist  attrs;
    Error_layer  layer;
    Error_severity severity = ERROR_RECOVERABLE;
    int n = 0;
    char strs[64][7];

    for (attrs = avlist; *attrs && n < 5; attrs = attr_next(attrs)) {
        switch ((int) attrs[0]) {
            case ERROR_BAD_ATTR:
                sprintf(strs[n++], "bad attribute %s",
                    attr_name(attr[1]));
                break;
            case ERROR_BAD_VALUE:
                sprintf(strs[n++],
                    "bad value (0x%x) for attribute %s", attrs[1],
                    attr_name(attrs[2]));
                break;
            case ERROR_INVALID_OBJECT:
                sprintf(strs[n++], "invalid object (%s)",
                    (char *) attrs[1]);
                break;
            case ERROR_STRING:
                sprintf(strs[n++], "%s", (char *) attrs[1]);
                break;
            case ERROR_PKG:
                sprintf(strs[n++], "Package: %s",
                    ((Xv_pkg *)attrs[1])->name);
                break;
            case ERROR_SEVERITY:
                severity = attrs[1];
        }
    }
    strcpy(strs[n++], "Core dump?");
    strs[n] = 0;
    if (notice_prompt(base_frame, (Event *)NULL,
        NOTICE_MESSAGE_STRINGS_ARRAY_PTR, strs,
        NOTICE_BUTTON_YES,               "Yes",
        NOTICE_BUTTON_NO,                "No",
        NULL) == NOTICE_YES)
            abort();
    if (severity == ERROR_NON_RECOVERABLE)
        exit(1);
    return XV_OK;
}
```

This error handling routine sets a set of error messages in an array of buffers. A notice is used to display the messages and prompt the user to generate a core image of the program for debugging. Selecting “Yes” causes `abort()` to be called. The program exits if the severity is non-recoverable, and continues otherwise.

One particular note of interest is the use of the routine `attr_name()`. This is a hypothetical routine that you would have to write to convert the actual enumerated attribute-values into strings that make sense to read. If you write your own XView package with new attributes, you will have to write a routine equivalent to `attr_name()`. The routine should return a static `char *` describing the attribute. If the attribute has no corresponding string (e.g., it is an unknown attribute), it should return a string indicating the integer or hexadecimal value.