

12

Setting Resources

This chapter describes how to set resource variables that determine application features such as color, geometry, fonts, and so on. It describes the syntax of resource definition files such as *.Xdefaults* and *.Xresources*, as well as the operation of *xrdb*, a client that can be used to change resource definitions dynamically and make resources available to clients running on other machines.

Before we go on to discuss resources in detail, we'll take a detour to talk about managing your environment in general. We do this to put the management of your X resources into the larger context of your total X environment.

Managing Your X Environment

This section discusses various tasks involved in managing your X user environment. In particular, it describes how to include common X client program directories in your search path and how to write a startup shell script. This discussion should be generally consistent with what you will see on your system, but specific details may vary among Linux distributions. See the documentation for your distribution for the specifics that apply to your system.

Including X in Your Search Path

X clients provided with X Windows are normally stored in the directory */usr/X11R6/bin*. If you should also see them in the directories */usr/bin/X11* or */usr/X11/bin*, just realize that these directories are probably symbolically linked to */usr/X11R6/bin*. In order to invoke them by name without specifying the whole path, you need to be sure that */usr/X11R6/bin* is in your search path. This is normally done from *.profile* if you are using the *bash* shell or *.cshrc* or *.tcshrc* if you are using *tcsh*, using a command similar to this:

```
bash:
export PATH=/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin
tcsh:
```

```
| set path=( /usr/local/bin /usr/bin /usr/X11R6/bin /bin ) |
```

The exact list of directories will differ from system to system (and may also include additional directories such as your home directory or local project directories).

A Startup Shell Script

You may want to set up your system to automatically configure and run certain clients every time you log in. The best way to do this is to create a script that runs the clients you want. Depending on how X is set up on your system, you can create and execute this script in one of two ways:

- If you log in through *xdm*, name the script *.xsession*, make it executable, and put it in your home directory. When you log in, *xdm* automatically executes your *.xsession* file. *.xsession* is generally a Bourne-shell script, but it can be any executable file. If *\$HOME/.xsession* doesn't exist, *xdm* gives you one *xterm* client and starts a window manager.
- If you start X either with *xinit* or with the *startx* shell script (which is a front-end to *xinit*), name your script *.xinitrc* and put it in your home directory. Both *xinit* and *startx* start the server and execute *.xinitrc*. Unlike *.xsession*, the *.xinitrc* script must be a Bourne-shell script and does not have to be executable. Among the functionality provided by *startx* is that if *\$HOME/.xinitrc* doesn't exist, *startx* defaults to the systemwide startup script */etc/X11/xinit/xinitrc*, whereas *xinit* just gives you a single *xterm* client.

Note that if you are running Red Hat Linux, it expects to find a file called *.Xclients* rather than *.xsession* or *.xinitrc*.

With some variation depending on the environment, set up your startup script to do the following:

- Run *xrdb* to load your resources file into the X server's resource database. Once you've loaded the resource file, you don't need to be concerned with setting resources unless you want to override a setting. The use of *xrdb* is explained in detail in the section "Setting Resources with *xrdb*" later in the chapter.
- Set up other server preferences, such as running *xmodmap* to customize the keyboard or running *xhosts* to allow other systems to access your display.
- Start any other clients you want on your default display, such as *xterm*, a web browser, a clock client like *xclock* or *oclock*, *xload*, etc.
- Start your window manager or desktop environment.

Example 12-1 shows a startup shell script that opens some windows on the display and then starts the *fvwm2* window manager. You can use this script as either an *.xsession* or *.xinitrc* script.

Example 12-1: Startup shell script

```
# !/bin/sh
# Get hostname
system=`hostname`

# If on a workstation, DISPLAY is set to :0.0 or unix:0.0. If you
```

```

# want to run remote clients, you need to use hostname:0.0. Set that
# up:
case $DISPLAY in
    unix:0.0|unix:0|:0.0|:0) REMOTEDISPLAY="$system:0.0";;
    *) REMOTEDISPLAY=$DISPLAY ;;
esac

# Load resource definitions from .Xresources
xrdb $HOME/.Xresources

# Set keyclick off and invoke the screen saver after
# seven minutes of idleness
xset c off s 420

# xconsole window will disappear if you do not log in at the console
xconsole -exitOnFail -geometry -0+0 &

# Now start up some xterms
# An xterm below xconsole
xterm -geometry -0+105 &
# Remote xterm in the lower-right corner
rsh ruby xterm -title RUBY -geometry +0-0 -display $REMOVEDISPLAY &

# Now start up other clients
# xcalc below the icon manager
xcalc -geometry +250+90 &

# Digital xclock below the icon manager
xclock -digital -update 1 -geometry +0+90 &

# xload on the bottom
xload -geometry -220-0 &

# Start the window manager in the foreground. Killing this window
# will shut down your X session.
# We've configured fvwm in the .fvwmrc file to bring up the Icon
# Manager automatically
fvwm2

```

One thing you'll notice in the startup script is that some commands are run in the background and some aren't. For clients that configure the X server, like *xrdb*, you want to be sure that the command completes before other clients are started (to make sure that the other clients retrieve the right resources) so you don't want it to run as a background process. Clients like *xterm* and *xclock*, however, remain active until they are explicitly terminated; they have to run in the background or they prevent the script from continuing.

The exception is the last command in the script, which should always start a window manager, which must run until it is explicitly terminated and must run as a foreground process. The X session only remains active as long as the startup script is active; thus you don't want the startup script to actually exit, because that would terminate the X session. The last command is often called the *controlling process*, since it is the only thing preventing the startup script and your X session from exiting. If you are using a desktop environment such as GNOME or KDE, or one of the newer window managers, your startup script may very well contain only the command to start the desktop. All the

particulars of your environment are set up either in resource files or in the desktop or window manager environment itself.

Figure 12-1 shows a display with windows set up by our example startup script. Note that the user has begun working in the two *xterm* windows. The local window shows the contents of his *.Xresources* file, and the remote *xterm*, running on the system ruby, shows the output of an *ls* command.

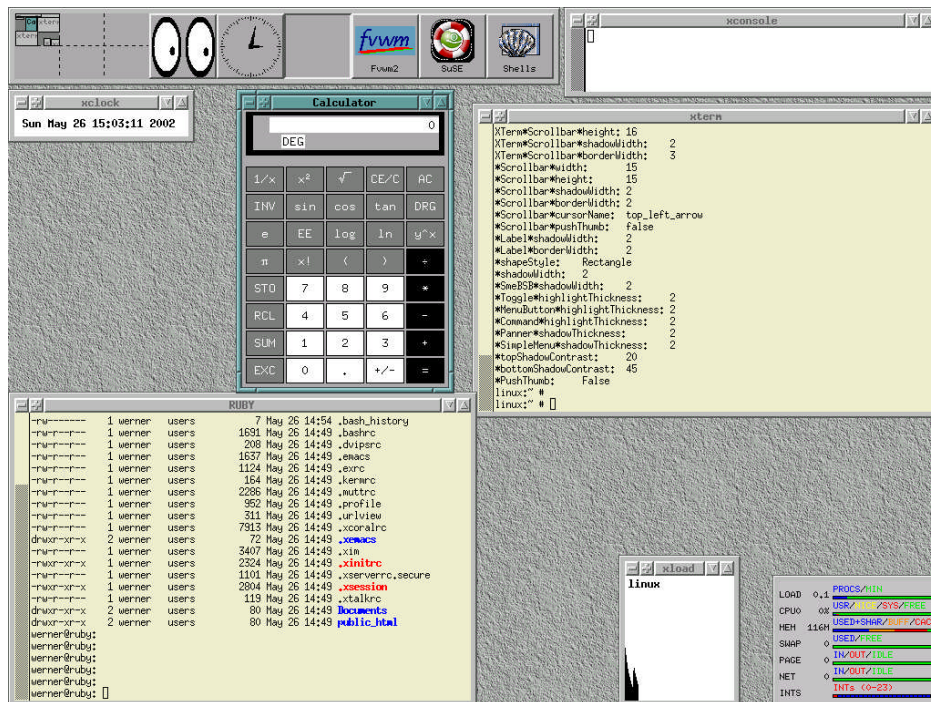


Figure 12-1: Display after running sample script

Note also that the windows we've created almost fully cover the display.* You might choose to run fewer (or smaller) windows and leave some room for additional clients, or you could place some of the windows on another virtual desktop and use the pager to access them. However, our sample display does leave room shuffling windows and also for invoking the root window menus. Create a display that works for you.

* This startup script was developed for and run on a 17" display with 1024x768 resolution. Differences in pixel sizes and resource definitions will probably make the coordinates and sizes of various windows come out differently on other hardware.

Customizing the X Environment

Virtually all X clients are customizable. You can specify how a client looks on the screen—its size and placement, its border and background color or pattern, whether the window has a scrollbar, and so on. Some applications even allow you to redefine the keystrokes or pointer actions used to control the application.

Traditional computer applications rely on command-line options to allow users to customize the way they work. As discussed in Chapter 2, *Getting Started Using X*, X applications also support command-line options, but often not for all features. In addition, there can be so many customizable features in an application that constantly entering a command line to set them all would be impractical.

While command-line options allow you to customize one instance of a client program, X also lets you specify characteristics that take effect every time you run a client. Almost every feature of a client program can be controlled using a variable called a *resource*. You can change the behavior or appearance of a program by changing the value associated with a resource. The following example shows six simple resource specifications for the *xclock* client. These particular resources specify the same characteristics as the command-line options we used to create the green and blue *xclock* in Chapter 2.

```
xclock*hands:    green
xclock*highlight: royalblue
xclock*background: lightblue
xclock*foreground: royalblue
xclock*update: 1
xclock*chime:    true
```

In many cases, a resource controls the same characteristic as a command-line option. However, setting a resource makes the characteristic the program default, while setting an option on the command line affects only that particular invocation of the client.

You generally place resource specifications in a file in your home directory. These can be in the file *.Xdefaults*, which is searched automatically by each client, or in a file called *.Xresources*. If you use *.Xresources*, you must run the client *xrdb*, the X resource database manager, to load the resources into a resource database that is kept in the X server and made available to all clients. We saw an example of this in the sample startup script earlier in the chapter. Note that if you use *xrdb*, you can actually give the file any name. It doesn't have to be called *.Xresources*, although that is the name usually used.

In addition to these resource files, which usually include your personal resource preferences for a number of client programs, many clients provide their own system-wide file containing application defaults that determine certain features for that client. For example, the system-wide *xterm* application default file contains resource definitions that specify such things as the labels for menu items, the fonts used to display menu items, and the shape of the pointer when it's in an *xterm* window. In addition to these default resource files, which are stored in a directory called */usr/X11R6/lib/X11/app-defaults*, you can create your own *app-defaults* directory in your home directory to override the defaults in the system *app-defaults* files for some clients. Finally, you can create resource files to set some resources only for the local machine, some for all machines on your network, and some for one or more specific machines.

Resources you set in your own resources file take precedence over system-wide default values. However, command-line options override both your own defaults and any system-wide defaults for a particular instance of the client process. It is important to note that command-line options normally take precedence over any prior resource settings. This way, you can set up the resource files to control the way you normally want your application to work, and then use command-line options to specify only the changes you want for a particular execution of the application. Within a single resource file, if a resource is specified more than once, the last specification is the one that applies.

The basic syntax of a resource specification looks complicated, but is actually fairly straightforward. A resource definition file, such as an *app-defaults* file or *.Xresources*, is basically a two-column list, where each line specifies a different resource. The simplest resource definition line has the name of the client, followed by an asterisk, and the name of the variable, followed by a colon, in the left column. The right column contains the value of the resource variable:

```
| client*variable: value
```

Each client recognizes certain resource variables that can be assigned a value. The resources you can set for a client are listed in its documentation. For example, here are some of the resources that are specified in */usr/X11R6/lib/X11/app-defaults/XCalc* for the *xcalc* client:*

```
| XCalc.Title:           Calculator
| XCalc.IconName:        Calc
| XCalc.IconPixmap:      calculator
| XCalc.Cursor:          hand2
| XCalc*Font:            8x13
| XCalc*ShapeStyle:      rectangle
| XCalc*Command.horizDistance:  2
| XCalc*Command.vertDistance:  2
| XCalc*Command.width:      40
| XCalc*Command.height:    26
| XCalc*Command.internalWidth: 1
```

Resource variables can take a value. For example:

```
| XCalc.Title:           Calculator
| XCalc*Command.width:    40
```

The first line sets the client's title to Calculator, and the second sets the Command widget's width to 40.

In other cases, resource variables may be Boolean, such as this example to turn on a scrollbar in an *xterm* window:

```
| xterm*scrollBar:      True
```

In applications written with the X Toolkit, resources may be associated with individual objects (called *widgets*) within an application. The syntax for specifying resources allows

* The name of the file in the *app-defaults* directory is that of the class to which the application belongs. Usually the first letter is capitalized, except for some applications whose name starts with x. Then it's often the first two letters that are used, for historical reasons. Hence *xcalc*'s *app-defaults* file is named *XCalc*.

for separate control over both a *class* of object in the application and an individual *instance* of an object. This is illustrated by these resource specifications for a hypothetical application called *xclient123*:

```
xclient123*Buttons.foreground:    blue
xclient123*help.foreground:      red
```

The first line sets the foreground color of all buttons (i.e., the class Buttons) in the *xclient123* application to blue; the second line overrides the first specification for one particular instance of the class Buttons, the help button, which is to have a red foreground.

Resource specifications can be much more complicated than we've seen so far. For applications written with the X Toolkit (i.e., most standard X clients), X allows you to specify different characteristics for individual widgets within the application. A client's graphical features such as menus, command buttons, dialog boxes, and scrollbars are made up of widgets. Most clients have a fairly complex widget hierarchy of widgets within widgets (e.g., a command button within a dialog box).

Resource-naming syntax parallels the widget hierarchy within an application. For example, you can set different background colors for different command buttons and specify still another background color for the dialog box that encloses them. In such cases, the actual widget names are used within the resource specification.

Some Common Resources

Every X Toolkit command-line option has a corresponding resource variable, and most X Toolkit applications recognize some subset of these resources. Table 12-1 lists the standard resource variables recognized by most X Toolkit clients.

Table 12-1: Common X Toolkit Resources

Instance Name	Default	Description
height	0	Height of window in pixels
width	0	Width of window in pixels
x	0	Window's x position in pixels
y	0	Window's y position in pixels
foreground	black	Foreground color
background	white	Background color
borderColor	black	Border color
borderWidth	1	Border width in pixels

Note that in a complex X Toolkit application, these values can occur at every level in a widget hierarchy. For example, our *xclient123* application might support these complete instance names:

```
xclient123.background
xclient123.buttonBox.background
xclient123.buttonBox.commandButton.background
```

```
| xcclient123.buttonBox.quit.background
```

These resources specify the background color for the application window, the buttonbox area, any command buttons, and the quit command button, respectively.

As we will see in the next section, the specification:

```
| xcclient123*background
```

matches all of these widgets.

Resource-Naming Syntax

The basic syntax of a resource definition file is fairly simple. Each client recognizes certain resource variables that can be assigned a value. Most of the common client programs are written to use the X Toolkit. Toolkits are a mechanism for simplifying the design and coding of applications and making them operate in a consistent way. Toolkits provide a standard set of objects, or widgets, such as menus, command buttons, dialog boxes, and scrollbars. As we'll see, the naming syntax for certain resources parallels the object hierarchy that is built into X Toolkit programs. In addition to certain application-specific resource variables, most clients that use the X Toolkit recognize the common set of standard X resource variables that we saw in Table 12-1.

As we've seen, the most basic entry you can have in a resource definition file consists of the name of a client, followed by a period or an asterisk, and the name of a variable. A colon and whitespace separate the client and variable names from the actual value of the resource variable. For example, the following line specifies that all instances of the *xterm* application have a scrollbar:

```
| xterm*scrollBar:      True
```

If you omit the name of the client, the variable applies to all instances of all clients that use that resource (in this case, all clients that can have a scrollbar). If you define the resource both for a particular client and for all clients, then the value of the client-specific variable takes precedence for that client. This is a good way to override a global specification for just one client, but still not have to specify it on the command line. So, for example, if you wanted for some reason to turn off scrollbars for all X clients except *xterm*, you could include these two lines in your resource file:

```
| *scrollBar:          False
| xterm*scrollBar:     True
```

Note that the global specification includes the asterisk before the name of the resource. Also, don't forget to include the colon; it's easy to forget it, and the resource manager provides no error messages. If there is an error in a resource specification, the specification is simply ignored and the value you set does not take effect.

Another common error is to include a space at the end of the line. The resource manager then searches for a value ending with a space. It fails to find this value but provides no error message and leaves you in the dark concerning the non-visible trailing space.

To include a comment in a resource file or to comment out a resource specification, begin the line with an exclamation point (!). For example:

```
| !-----
```

```
| ! emacs defaults
| !-----
| !emacs*geometry:      80x50
| emacs*geometry:      80x40
```

In this example, the first three lines are strictly comments. The first **emacs*geometry** entry is a geometry resource specification that has been commented out. That specification has been replaced by another geometry specification that sets up an 80x40-line window instead of the original 80x50-line window.

If the last character on a line is a backslash (\), the resource definition on that line is assumed to continue on the next line. See the section “Event Translations” for examples using the backslash.

Syntax of Toolkit Client Resources

As described above, X Toolkit applications (and Xt-based toolkit applications) are made up of predefined components called widgets. Widgets can contain other widgets (e.g., a dialog box can contain a command button). The syntax for the resource specification of Toolkit clients parallels the levels of the widget hierarchy. Think of a resource specification as having this format:

```
| object.subobject[.subobject...].attribute: value
```

The parts of the specification are:

object

The client program or a specific instance of the program.

subobjects

The levels of the widget hierarchy (usually, the major structures within an application, such as windows, menus, scrollbars, etc.).

attribute

A feature of the final *subobject*, such as the background color or a label that appears on the subobject to which the value applies.

value

The actual value that is to be set for the resource *attribute*, i.e., the label text, color, or other feature.

The type of value to supply is often evident from the name of the resource. In most cases, the possible values are the same values you would use with the corresponding command-line option.

For example, various resources, such as **borderColor** or **background**, take color specifications; **geometry** takes a geometry string, **font** takes a font name, and so on. Boolean values, such as those taken by **scrollBar**, can generally be specified as: **on** or **off**; **yes** or **no**; or **True** or **False**.

Tight and Loose Bindings

Binding refers to the way in which components of a resource specification are linked together. Resource components can be linked in two ways:

A *tight* binding

Represented by a period (.). The components on either side of the period must be next to one another in the widget hierarchy. Specifying a tight binding requires you to know the exact widget hierarchy.

A *loose* binding

Represented by an asterisk (*). The asterisk is a wildcard character that means there can be any number of levels in the hierarchy between the two surrounding components (including none). Using a loose binding is the most common way to specify a resource because it doesn't require knowing the details of the hierarchy.

If you want to specify tight bindings, you must be very familiar with the widget hierarchy: it's easy to use tight bindings incorrectly. For example, the following resource specification to request that *xterm* windows be created with a scrollbar does *not* work:

```
# This does not work!
xterm.scrollBar:    True
```

The specification is wrong because in the *xterm* widget hierarchy, the VT102/220 window is considered to be one widget, the Tektronix window another, and the menus a third widget. This means that in order to use tight bindings to request that *xterm* windows be created with a scrollbar, you would need to specify the particular widget, for example:

```
xterm.vt100.scrollBar:    True
```

But it is far simpler, and more common, to use the asterisk connector:

```
xterm*scrollBar:    True
```

Note that the asterisk is both similar to and different than resource syntax in the shell. In the shell the asterisk is a wildcard that can represent zero or more characters; in a resource file, it is a wildcard that represents zero or more missing components of the resource name. However, unlike the shell, where an asterisk can match partial filenames, you cannot use the asterisk in a resource file to match partial component names. If you want to set the same specification for clients with similar names, you cannot use a common abbreviation. For example, you *cannot* set *xcalc*, *xclock*, and *xclipboard* to all display in reverse video (i.e., with foreground and background colors swapped) with the single entry:

```
# This does not work!
xc*reverseVideo:    True
```

In an application that supports multiple levels of widgets, you can mix asterisks and periods. In general, though, it's safer to use asterisks as the connector even with simple applications, so that even if a new release of an application adds new levels to the hierarchy, your resource settings will still work.

Instances and Classes

Each component of a resource specification has an associated *class*. Different widgets or widget attributes may have the same class. For example, in the case of *xterm*, the color of text (specified as the resource variable *foreground*), the pointer color, and the text cursor color are all defined as instances of the class *Foreground*. This makes it possible to set the value of all three with a single resource specification. That is, if you want to make the text, the pointer, and the cursor dark blue, you can specify either:

```
| xterm*foreground:      darkblue
| xterm*cursorColor:    darkblue
| xterm*pointerColor:   darkblue
```

or you can achieve the same thing with a single line by just referring to their class:

```
| xterm*Foreground:     darkblue
```

Initial capitalization is used to distinguish class names from instance names. By convention, class names always begin with an uppercase letter, while instance names always begin with a lowercase letter. Note, however, that if an instance name is a compound word (such as **cursorColor**), the second word is usually capitalized. It is only the case of the initial character that is significant.

The real power of class and instance naming is not apparent in applications such as *xterm* that have a simple widget hierarchy. However, in complex applications, class and instance naming allow you to do things such as specify that all buttons in a dialog box be blue except for one particular button that is to be red. For example, going back to our hypothetical *xclient123* application, you might have a resource file that reads:

```
| xclient123*buttonbox*Buttons*foreground:  blue
| xclient123*buttonbox*delete*foreground:   red
```

where **Buttons** is a class name and the **delete** button is an instance of the **Buttons** class. This type of specification works because an instance name always overrides the corresponding class name for that instance. Class names thus allow default values to be specified for all instances of a given type of object, while instance names can be used to specify exceptions to the rules outlined by the class names.

A class name can be used with a loose binding to specify a resource for all clients. For example, the following entry specifies that the foreground color for all clients should be blue:

```
| *Foreground:         blue
```

The documentation for a given program should* always give you both instance and class names for every resource variable you can set. You'll notice that in many cases the class name is identical to the instance name, with the exception of the initial capital letter. Often (but not always) this means that there is only one instance of that class. In other cases, the instance with the same name as the class is simply the primary or most obvious instance of the class.

Using ? as a Wildcard

You can use a question mark (?) to represent *any single component* in a resource specification except the final component, which is the resource variable itself. For example:

```
| xclient123.?.?.Background:  whitesmoke
```

* Though the documentation should be informative, this is regrettably not always the case. The section "Testing and Editing Resources with editres" presents the *editres* tool, which can help you find a client's undocumented resources.

This specification sets the background color for all widgets that are two subobjects below the application level. You can think of these subobjects/widgets as the *grandchildren* of the top-level window in the client's window hierarchy. Typical grandchildren might be dialog boxes, menus, etc.

Note that this specification sets the background color for *only* those widgets--the tight bindings ensure this. Setting a loose binding between the second question-mark wildcard and the resource variable (**Background**) would expand the coverage to include the second subobject level and also all further subobjects:

```
| xclient123.?.?*Background:    whitesmoke
```

The use of the question mark requires a bit of finesse, but it simplifies specifications that would otherwise be very involved.

There is an important distinction between the use of the question mark and the asterisk wildcards. The question mark always represents a single component and therefore must always be bracketed by connectors (usually periods) unless it is the first component. The presence of a question mark specifies that a component exists, though it does not specify the name of the component.

The asterisk, on the other hand, does not tell you anything about how many components have been omitted, just that zero or more components are not specified. The only requirement is that the missing components must be adjacent to each other in the widget hierarchy.

Precedence Rules for Resource Specification

Even within a single resource file, resource specifications often conflict. For instance, let's go back to our earlier example involving the hypothetical *xclient123* application:

```
| xclient123*Buttons.foreground:    blue
| xclient123*help.foreground:      red
```

The first resource specification makes the foreground color of all buttons in the class **Buttons** blue. The second resource specification overrides the first in one instance; it makes the foreground color of the **help** button, which is an instance of the class **Buttons**, red. In the event of conflicting specifications, there are a number of rules that the resource manager follows in deciding which resource specification should take effect.

We've already seen two of these rules:

- Instance names take precedence over class names.
- Tight bindings take precedence over loose bindings.

From just these two rules, we can deduce a general principle: the more specific a resource definition is, the more likely it is to be honored in the case of a conflict.

However, for cases in which you want to set things up very carefully, you should know a bit more about how programs interpret resource specifications.

For each resource, the program has both a complete, fully specified, tightly bound instance name and a class name. In evaluating ambiguous specifications, the program compares the specification against both the full instance name and the full class name. If a

component in the resource specification matches either name, it is accepted. If it matches more than one element in either name, it is evaluated according to these precedence rules:

1. The levels in the hierarchy specified by the user must match the program's expectations or the entry is ignored. For example, if the program expects either the instance name, `xterm`:

```
| xterm.vt100.scrollBar: value
```

or the class name, `XTerm`:

```
| XTerm.VT100.ScrollBar: value
```

then the resource specification:

```
| xterm.scrollBar: True
```

does not work, because the tight binding is incorrect. The objects `xterm` and `scrollBar` are not adjacent in the widget hierarchy: there is another widget, `vt100`, between them. However, the specification works if you use a loose binding:

```
| xterm*scrollBar: True
```

Or you can use the question-mark wildcard to represent `vt100` in the widget hierarchy:

```
| xterm.?.scrollBar: True
```

This specification is perfectly valid. Note also that this line is more specific than (and thus takes precedence over) `xterm*scrollBar: True`.

2. Tight bindings take precedence over loose bindings. That is, entries with instance or class names prefixed by a period are more specific than entries with names prefixed by an asterisk, and more specific entries take precedence. For example, `xterm.vt100.geometry` takes precedence over `xterm*geometry`.
3. Similarly, instances take precedence over classes. For example, the instance `*scrollBar` takes precedence over the class `*Scrollbar`.
4. Components to the left carry more weight than components to the right. Thus `xterm*background` takes precedence over `*background`.
5. An instance or class name that is explicitly stated takes precedence over one represented by the question-mark wildcard, which in turn takes precedence over an omitted component represented by an asterisk. Thus, `xterm*scrollbar` is more specific than `?*scrollBar`, which is still more specific than `*scrollBar`.

To illustrate these rules, let's consider the following resource specifications for our `xclient123` application:

```
| xclient123.toc*Command.activeForeground: black
| *Command.Foreground: green
```

Both lines specify a foreground color; both specifications are valid. Now, applying the rules of precedence, let's determine what foreground color will be used for the `xclient123` application's `include` command button. To determine how conflicting specifications are applied, the program tries to match these specifications against the complete tightly bound instance and class specifications. In this case, say the complete specifications are:

* Note that `xterm`'s class name is `XTerm`, not `Xterm` as you might expect.

<code>xclient123.toc.messageFunctions.include.foreground</code>	<i>instance name</i>
<code>Xclient123.Box.SubBox.Command.Foreground</code>	<i>class name</i>

Note that these specifications are the instance and class names for the same resource, which determines the foreground color of the **include** button. Each component of the instance name belongs to the class in the corresponding component of the class name. Thus, the instance **toc** occurs in the class **Box**, the **messageFunctions** instance name is from the class **SubBox**, etc. The **include** button is an instance of the **Command** class.

Both resource specifications match these instance and class names. However, with its tight bindings and instance names, `xclient123.toc*Command.foreground` matches more explicitly (i.e., with higher precedence). The specification `*Command.Foreground` also matches the instance and class names, but it is composed entirely of less-specific class names and thus has lower precedence. Therefore, the resource is set so that the **foreground** color of the **include** button is **black**.

However, since the second line is also an acceptable specification, it sets the foreground color of other objects in the **Command** class. Thus, if there are other *xclient123* command buttons comparable to the **include** button in the hierarchy, this line sets the foreground color of these buttons to **green**. In addition, since the line begins with an asterisk, the resource is set not just for *xclient123*, but also for any other application with a **Command** class.

Now let's consider some actual conflicting resource specifications and apply the rules of precedence. All three of the resources in the following example are valid font specifications for all instances of the class **Command** (i.e., for the font to be used on command-button labels).

<code>*Command*Font: -*-helvetica-bold-r-normal--*-120-*-*--iso8859-1</code>
<code>?*Command*Font: 7x14</code>
<code>XClipboard*Command*Font: 6x10</code>

The resources are listed in increasing order of specificity. Because of the initial loose binding, the first specification applies to any client with a command widget. The second specification also applies to any client with a command widget, but the initial question mark (representing all clients) makes it more specific. Thus, the second line overrides the first and specifies that the font for command buttons for all applications is **7x14**.

The third line is even more specific because it begins with an actual class name (**XClipboard**) rather than the question-mark wildcard. This line specifies that the font for *xclipboard* command buttons is **6x10**. Note, however, that the second line still applies to all other clients--they will use the font **7x14** for command buttons--the third line simply introduces an exception.

Event Translations

From the examples in previous sections, you can see that the purpose of many resource variables is frequently self-evident through their choice of names. There are also some less-obvious resource variables; among these is one type of specification, an *event translation* that can be used with many clients.

User input and other information passes from the server to a client application in the form of *events*. An event is a packet of information that tells the client something it needs to act on, such as keyboard input. As mentioned in Chapter 1, *Introduction*, moving or clicking the pointer or pressing a key causes input events to occur. When a program receives a meaningful event, it responds with some sort of action.

For many clients, the resource manager recognizes mappings between certain input events (such as a pointer-button click) and some sort of action by the client program (such as selecting text). A mapping between one or more events and an action is called a *translation*. A resource containing a list of translations is called a *translation table*.

The Syntax of Event Translations

The operation of many clients, notably *xterm*, is partly determined by default input event translations. Let's look, for example, at text selection. When you select text with the first pointer button (an event), the text is saved into memory (an action).

In this case, the input "event" is actually a sequence of three separate X events:

1. Pressing the first pointer button unselects any previously selected text and begins selection of new text.
2. Moving the pointer while holding down the first button extends the selection.
3. Releasing the button ends the selection and saves the text into memory (both as the PRIMARY selection and into CUT_BUFFER0).

The event and action mappings can be expressed in a translation table as:

```
<Btn1Down>: select-start()\n\
<Btn1Motion>: select-extend()\n\
<Btn1Up>: select-end(PRIMARY,CUT_BUFFER0)
```

Each event is enclosed in angle brackets (<>) and produces the action that follows the colon (:). A space or tab generally precedes the action, though this is not mandatory:

```
| <event>: action
```

A translation table must be a continuous string. In order to link multiple mappings as a continuous string, each event-action line is terminated by a newline character (`\n`) followed by a backslash (`\`) to escape the actual newline.

The mappings shown above are simplified versions of the default *xterm* translations. You can see the actual default translations in the *xterm* manpage if you are interested.* All the events are simple, comprised of a single button motion. Events can also have modifiers: i.e., additional button motions or keystrokes (often the CTRL or Meta keys) that must be performed with the primary event to produce the action. (Events can also have modifiers that *must not* accompany the primary event if the action is to take place.)

You can specify non-default translations using a translation table. Since actions are part of the client application and cannot be modified, what you are actually doing is specifying

* In general, the event-action mappings that can be modified for a particular client using translation resources are described in the client's documentation.

alternative events to perform an action.* Keep in mind that only applications written with the X Toolkit (or another Xt-based toolkit) recognize translation table syntax.

The basic syntax for specifying a translation table as a resource is:

```
[object*[subobject...]]*translations:  #override\  
[modifier]event: action
```

The first line looks a lot like a standard resource specification. However, there are a few differences. First, the final argument is always **translations**, indicating that one (or more) of the event-action bindings associated with the **[object [subobject...]]** is being modified.

Second, **#override** is not the value of the resource; it is literal and indicates that what follows overrides any default translations.

Finally, the translation-table equivalent of a resource value is a single event-action mapping or a list of several mappings linked by the characters **\n** to make the resource a continuous string. The **#override** is also followed by a backslash (****) to make the next line a part of the string.

A not-so-obvious principle behind overriding translations is that you only literally “override” a default translation when the event(s) of the new translation exactly match the event(s) of the default translation. If the new translation does not conflict with any existing translation, it is appended to the defaults.

The following **xterm** translation table shows multiple event-action mappings linked in this manner:

```
XTerm.VT100.Translations: #override \  
  <Btn1Down>:  select-start() \n\  
  <Btn1Motion>: select-extend()\n\  
  <Btn1Up>:    select-end(PRIMARY, CUT_BUFFER0)
```

xterm translations to use xclipboard

Let’s look at an example where you might want to make use of translations. The **xclipboard** client provides a window in which you can store text selected from other windows. You can also paste text from the **xclipboard** window into other windows. You can cut and paste between many different X clients, but for now let’s assume that we’re primarily interested in cutting and pasting text between **xterm** windows.

You can specify translations for **xterm** so that text you copy with the pointer is made the **CLIPBOARD** selection. The **CLIPBOARD** selection is the property of the **xclipboard** client; once text has been copied to the **CLIPBOARD**, it automatically appears in the **xclipboard** window.

Some sample translations that allow you to use the **xclipboard** in this way are:

```
XTerm.VT100.Translations: #override \  
  Button1 <Btn3Down>: select-end(PRIMARY,CUT_BUFFER0,CLIPBOARD) \n\  
  !Shift <Btn2Up>:    insert-selection(CLIPBOARD) \n\  
  ~Shift ~Ctrl ~Meta <Btn2Up>: insert-selection(PRIMARY,CUT_BUFFER0)
```

* You’ll see later that in certain cases you may be able to supply an alternative **argument** (such as a selection name) to an action. In that case the argument is interpreted by the resource manager.

These translations include a feature we haven't discussed yet--the use of *modifiers*. As mentioned earlier, events can have modifiers, additional button motions or keystrokes that are performed with the primary event. According to the first line of this translation table, selecting text with **Button1** (the modifier) during the event of pressing the third pointer button (**Btn3Down**) produces the action of making the text the **CLIPBOARD** selection (as well as making it the **PRIMARY** selection and saving it to **CUT_BUFFER0**). Basically, we've taken the default **select-end** translation--which uses the first pointer button and the arguments **PRIMARY** and **CUT_BUFFER0**--and added the **Btn3Down** action and the **CLIPBOARD** argument.

The second line specifies that you paste the **CLIPBOARD** selection into an *xterm* window by holding the Shift key and clicking the second pointer button in that window. The exclamation point (!) before **Shift** means that the Shift key is the only modifier allowed. This is in contrast to the previous line, where **Button1** is the modifier, but another modifier could also be pressed and the translation would still work.

The third line modifies the way the contents of the **PRIMARY** selection or **CUT_BUFFER0** are pasted into a window. By default, pressing the second pointer button pastes the contents of the **PRIMARY** selection. If there is no **PRIMARY** selection, the contents of the cut buffer are pasted. The default translation that sets this behavior is the following:

```
| ~Ctrl ~Meta <Btn2Up>:    insert-selection(PRIMARY,CUT_BUFFER0)
```

This translation specifies that clicking (actually releasing) pointer button 2 while pressing any modifier button or key other than CTRL or Meta performs the **insert-selection** action of inserting text. (The tilde (~) is the "not" modifier.) The default excludes the CTRL and Meta keys to prevent conflict with other action mappings. Adding **~Shift** to the default action prevents a conflict with the action that pastes the **CLIPBOARD** selection.

This translation does not work with a two-button mouse using three-button emulation because simultaneously pressing the first and third pointer buttons is treated as a button-two press.

According to the translations in this example, if you select text as usual with the first pointer button and then press the third button while continuing to hold down the first button, the text becomes the **CLIPBOARD** selection and appears automatically in the *xclipboard* window.

Notice that we haven't modified the way text is selected; that is, the first translation is a different event/action mapping than the one that specifies text selection, so the default translation still applies. If you select text with the first pointer button alone, that text is still made the **PRIMARY** selection and fills **CUT_BUFFER0**. To send text to *xclipboard*, you also have to press the third pointer button. Thus, not all selected text needs to be sent to the *xclipboard* window--chances are you don't want to save every piece of text you copy.

Using translation tables to map commands to function keys

The sample *xterm* translations to use the clipboard involve just a few of the actions *xterm* recognizes. Among the more useful translations you can specify for *xterm* are function-key mappings that let you to enter frequently used commands with a single keystroke.

This sort of mapping involves an action called `string`, which passes a text string to the shell running in the `xterm` window.

The translation-table syntax for mapping function keys is straightforward. For example, the following line maps the text string `df -k` (which reports filesystem disk usage in kilobytes) to the F1 function key:

```
| <Key>F1:      string("df -k")
```

Notice that the text string is enclosed in quotes. Any time the argument to `string` includes spaces or non-alphanumeric characters, as it does in this example, the whole argument must be enclosed in a pair of double quotes.

The translation table for this key mapping looks like this:

```
| XTerm.VT100.Translations: #override \  
|   <Key>F1: string("df -k")
```

This translation causes `df -k` to be passed to the command line in the active `xterm` window when you press the F1 key.

Notice, however, that the command is not invoked because no carriage return is specified in the sample translation. You can add a return as the argument to another `string` action within the same translation.

To specify another key, use the hexadecimal code for that key as the argument to `string`. Keycodes and the procedure for determining them are explained in Chapter 3, *A Selection of Useful X Clients*. You can get a list of all keycodes with the command `xmodmap -pk`. To find the keycode for the Return key, enter the command:

```
| xmodmap -pk | grep Return
```

This produces the following:

```
| 36          0xff0d (Return)
```

In this case, the hexadecimal code is `0xff0d`. To enter a keycode as an argument to `string`, you need the `0x` prefix to indicate that the code is hexadecimal, followed by the specific code, which in this case is `0d`. You can omit the `ff`.^{*} Putting this all together, we can specify the following translation table to map the F1 key to the sequence `df -k` followed by RETURN for an `xterm` window:

```
| XTerm.VT100.Translations: #override \  
|   <Key>F1: string("df -k") string("0x0d")
```

Now if you press F1 in an `xterm` window, the command is not just entered on the command line but is executed as well, as shown in Figure 12-2.

^{*} For some codes, the `ff` is replaced by `00`, but all the hexadecimal keycodes include one or the other.

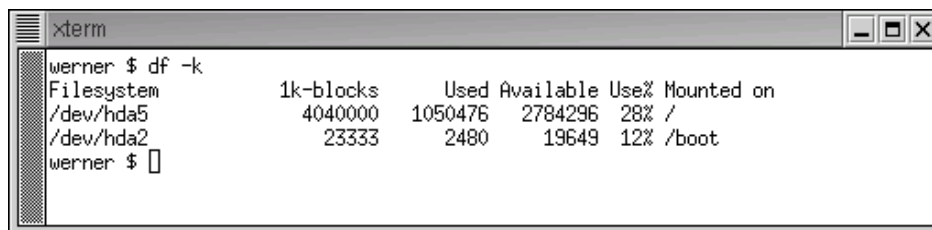


Figure 12-2: Pushing F1 passes command text to xterm shell

Remembering that we can list several translations in a single table, we can go on to map some additional function keys. The following table maps function keys F1 through F3:

```
XTerm.T100.Translations: #override \
<Key>F1: string("df -k") string(0x0d) \n\
<Key>F2: string("cd ~/bitmap;ls") string(0x0d) \n\
<Key>F3: string("cd /usr/X11R6/lib/X11") string(0x0d)
```

According to these translations, pressing F2 inserts the command string `cd ~/bitmap;ls`, which changes the directory to `~/bitmap` and then lists the contents of that directory. Note that you can issue multiple commands (`cd`, `ls`) with a single key by separating them with a semicolon (`;`) just as you would on the command line. Pressing F3 changes the directory to `/usr/X11R6/lib/X11`.

All the translations for one application can appear in the same table. For example, we can combine the `xterm` translations for the clipboard with the translations to map function keys:

```
XTerm.VT100.Translations: #override \
Button1 <Btn3Down>: select-end(PRIMARY,CUT_BUFFER0,CLIPBOARD) \n\
!Shift <Btn2Up>: insert-selection(CLIPBOARD) \n\
~Shift ~Ctrl ~Meta <Btn2Up>: insert-selection(PRIMARY,CUT_BUFFER0)
<Key>F1: string("df -k") string(0x0d) \n\
<Key>F2: string("cd ~/bitmap;ls") string(0x0d) \n\
<Key>F3: string("cd /usr/X11R6/lib/X11") string(0x0d)
```

The order of the translations is not important. However, you do need to end all but the final line with the sequence `\n\` to make the resource a continuous string.

Setting Resources

Learning to write resource specifications is a manageable task once you understand the basic rules of syntax and precedence. In contrast, the multiple ways you can set resources--for a single system, for multiple systems, for a single user, for all users--can be confusing. For purposes of this chapter, we're primarily concerned with specifying resources for a single user running applications both locally and remotely.

As we've seen, resources are generally specified in files and loaded into the X server by the `xrdb` client. `xrdb` is normally run from your startup file or run automatically by `xdm` when you log in. Even if no resource file is loaded by `xrdb`, the `.Xdefaults` file (if it exists), and hence its resources, is always read by an X application.

Remember that X allows clients to run on different machines across a network, not just on the machine that supports the X server. The problem with the older `.Xdefaults` mechanism is that users who run clients on multiple machines must maintain multiple `.Xdefaults` files,

one on each machine. By contrast, *xrdb* stores the application resources directly in the server, thus making them available to all clients, regardless of the machine on which the clients are running. As we'll see, *xrdb* also allows you to change resources without editing files.

Of course, you may want certain resources to be set on all machines and others to be set only on particular machines. See the section “Other Sources of Resource Definitions” later in this chapter for information on setting machine-specific resources.

A Sample Resources File

Example 12-2 shows a sample resources file. This file sets the border width for all clients to a default value of two pixels, and sets some other variables for *xclock* and *xterm*. The meaning of each variable is generally obvious from its name (for example, *xterm*scrollBar: True* means that *xterm* windows should be created with a scrollbar). As mentioned earlier, comment lines begin with an exclamation point (!).

For a detailed description of every possible variable, see the appropriate client documentation.

Example 12-2: A sample resources file

```
*borderWidth:      2
!
! xclock resources
!
xclock*borderWidth: 5
xclock*geometry:    64x64
!
! xterm resources
!
xterm*curses:       on
xterm*cursorColor:  skyblue
xterm*pointerShape: pirate
xterm*jumpScroll:   on
xterm*saveLines:    300
xterm*scrollBar:     True
xterm*scrollKey:     on
xterm*background:   black
xterm*borderColor:  blue
xterm*borderWidth:  3
xterm*foreground:   white
xterm*font:          8x13
```

Note that this sample resources file, which could be an *.Xresources* or *.Xdefaults* file, differs from the *app-defaults* file we saw earlier in the section “Customizing the X Environment” because it includes resources for multiple client applications. It also differs from a startup file (*.xsession* or *.xinitrc*) because it doesn’t actually run any clients; it just specifies how the clients appear and how they behave when they are run.

Specifying Resources From the Command Line

Two command-line options supported by all clients written with the X Toolkit can be useful in specifying resources.

The `-xrm` option

The `-xrm` option allows you to set on the command line any specification that you would otherwise put into a resources file. For example:

```
| xterm -xrm 'xterm*Foreground: blue' &
```

When you specify a resource on the command line, you must put single quotes around it, as shown in this example. The `-xrm` option specifies the resource(s) only for the current instance of the application. Resources specified in this way do not become part of the resource database.

The `-xrm` option is most useful for setting class resources. Since most clients already have command-line options that correspond to instance-variable resources, you don't need to use `-xrm` because you can just use the command-line option. For example, you can use the `-fg` command-line option to set the `foreground` color for a window, but you need to use `-xrm` to set the class `Foreground`.

Also, note that a resource specified with `-xrm` does not take effect if a resource that takes precedence has already been loaded with `xrdb`. For example, say you've loaded a resource file that includes the specification:

```
| xterm*pointerShape: pirate
```

If you then try to specify another cursor on the command line, that specification will fail:

```
| xterm -xrm '*pointerShape: gumby' &
```

This specification doesn't work because the resource `xterm*pointerShape` is more specific than `*pointerShape`. As a result, your `xterm` will have a pirate cursor instead of Gumby.

To override the resource database and get the Gumby cursor, you need to use a resource at least as specific, such as the following:

```
| xterm -xrm 'xterm*pointerShape: gumby' &
```

How `-name` affects resources

In X, the name of an application affects how resources are interpreted; you can use this fact to specify resources selectively for various instances of a client, based on the contents of the `-name` option. This option lets you assign a name to an instance of an application. Once you've assigned a name, the server identifies that particular instance of the application by that name.

For example, the following command sets the name of an instance of `xterm` to `bigxterm`:

```
| xterm -name bigxterm &
```

Once that `xterm` window is running, the `xterm` client uses any resources specified for `bigxterm` rather than for `xterm` for that particular `xterm` window. Thus you can create different instances of an application that use different resources. For example, you can put the following entries into your resource file:

```
| XTerm*Font:      8x13
| smallxterm*Font: 6x10
| smallxterm*Geometry: 80x10
| bigxterm*Font:   9x15
```

```
| bigxterm*Geometry: 80x55
```

You can then use the following commands to run *xterm* windows with different specifications:

```
| xterm &
```

creates an *xterm* with the default specifications, while:

```
| xterm -name bigxterm &
```

creates a big *xterm*, 80 characters across by 55 lines down, with a fontsize of **9x15**. The command:

```
| xterm -name smallxterm &
```

creates a small *xterm*, 80 characters across by 10 lines down, with a fontsize of **6x10**.

You can, of course, start up more than one *xterm* of any size; for example, you might want to run three instances of *bigxterm* and one instance of *smallxterm*.

Setting Resources with xrdp

The *xrdp* program stores resources in the X server, making them available to all clients, running on any system that accesses the server. *xrdp* saves you from maintaining multiple resource files if you run clients on more than one machine. (Technically speaking, the values of resource variables are stored in a data structure referred to as the **RESOURCE_MANAGER** property of the root window of screen 0 for the server.)

The appropriate *xrdp* command line is normally placed in your *.xinitrc* file or *.xsession* file to initialize your resources when you log in. However, it can also be invoked interactively if you want to modify your resource definitions once X is running. *xrdp* has the following syntax:

```
| xrdp [options] [filename]
```

The *xrdp* client takes a number of options, all of which are documented in its manpage. Some of the most useful options are discussed here and in the next few sections.

The optional *filename* argument specifies the name of a file from which the values of client variables (resources) are read. Normally, you'd specify the name of your resource file. For example:

```
| xrdp -load $HOME/.Xresources
```

or simply

```
| xrdp $HOME/.Xresources
```

to load the file *.Xresources* from your home directory. If you don't specify a filename, *xrdp* expects to read its data from standard input and waits for you to type some data, followed by an end-of-file character (usually CTRL-D). Note that whatever you type overrides the previous contents of the resource database, so if you inadvertently type *xrdp* without a filename argument and then quit with CTRL-D, you will delete any existing resource values. (If that happens, you can append new settings by running *xrdp* again with the **-merge** option discussed later in this section.)

The `-load` option is the default, which loads the contents of the resource file, in this case `$HOME/.Xresources`, into the resource database.

Querying the Resource Database

You can find out what resources are currently set by running `xrdb` with the `-query` option. For example:

```
xrdb -query
xterm*ScrollBar:    True
xterm*borderWidth:  3
xclock*foreground:  royalblue
*background:        #ffffff
```

If your initialization files are not set up to run `xrdb` when you log in, and you have not yet run it manually, you won't see any output when you query the database.

Using `appres` to list resources for a client

Another way of querying the resource database is to use the `appres` client. Rather than showing all resources in the database, `appres` shows the resources currently set for a particular client or hierarchy within a client. `appres` can be run with optional parameters that print the resources seen by an application of the specified `classname` or `instancename`. For example to see which resources are valid for all clients:

```
appres
*background:        #ffffff
```

Or to see which clients are valid only for `xterm`:

```
appres xterm
xterm*ScrollBar:    True
xterm*borderWidth:  3
*background:        #ffffff
```

Notice that the instance `background` appears in both cases. This is because `appres` doesn't really know anything about classes and instances as they may be defined by the client itself. It simply loads the resource database into a temporary file and does a string comparison:

```
[*.]
classname[*.]
instancename[*.]
```

and then prints out the lines that match. Note that `[*.]` means either `*` or `.`.

Loading New Resource Values

By default, when you run `xrdb`, it reads the input and stores the results into the resource database, replacing any previous values. If you simply want to add one or more new values to the database, you can use the `-merge` option. This option merges new values into the database; existing variables are preserved rather than overwritten with empty values. Existing variables are only overwritten if they exist in the input file to be merged.

For example, let's say you want to add new resources, which you have stored in the file `new.values`. You can say:

```
xrdb -merge new.values
```

As another example, if you want all new *xterm* windows to have scrollbars, you can use standard input and enter:

```
$ xrdp -merge
xterm*scrollBar:    True
CTRL-D
```

If you have problems getting your specifications to work, you may be having precedence problems. Use the *-query* option to list the values in the resource database and look for conflicting specifications.

Note that adding new specifications affects only new programs; it doesn't affect any programs that are already running. This is true even if you overwrite the existing specifications by loading a new resource file. Only programs initiated *after* you've run *xrdp* reflect the new settings.

Saving Active Resource Definitions in a File

Assume that you've loaded the resource database from an *.Xresources* or other file. However, you've modified the database using the *-merge* option and you'd like to make the new version your default.

Though possible, you don't need to edit the resource file manually. The *-edit* option to *xrdp* allows you to write the current value of the resource database to a file. If the file already exists, it is overwritten with the new values. However, *xrdp* is smart enough to preserve any comments and preprocessor declarations in the file being overwritten, replacing only the resource definitions.

For example:

```
| xrdp -edit ~/.Xresources
```

saves the current contents of the resource database in the file *.Xresources* in your home directory.

If you want to save a backup copy of an existing file, add the *-backup* option:

```
| xrdp -edit .mydefaults -backup old
```

The string following the *-backup* option is used as an extension to be appended to the old filename. In this example, the previous copy of *.mydefaults* is saved as *.mydefaults.old*.

Removing Resource Definitions

You can delete the definition of the resource database from the server by calling *xrdp* with the *-remove* option.

The only way to delete an individual resource definition from the database is to read the current *xrdp* values into a file, edit the file, and load the new values by running *xrdp* again using the edited file as input.

Other Sources of Resource Definitions

If *xrdb* has not been run, the **RESOURCE_MANAGER** property is not set and the resource database is empty. Instead, the resource manager looks for a file called *.Xdefaults* in the user's home directory. Resources found in this way are only available to clients running on the local machine.

Whether or not resources have been loaded with *xrdb*, when a client is run the sources of resource definitions are consulted in this order:

1. The client's application-defaults file (if any), which usually resides in the directory */usr/X11R6/lib/X11/app-defaults*, is loaded into the resource manager. Application-specific resource files generally have the name *Class*, where *Class* is the class name of the client program. For example, */usr/X11R6/lib/X11/app-defaults/XCalc* is the application defaults file for the *xcalc* client, which has a class name of XCalc.
Any other application-specific resource files: a resource file named by the environment variable `XUSERFILESEARCHPATH`; or if this variable is not set, a file in the directory named by the environment variable `XAPPLRESDIR`; otherwise, a file in the user's home directory.
2. Resources loaded into the **RESOURCE_MANAGER** property of the root window with *xrdb*; these resources are accessible regardless of the machine on which the client is running.
3. Resources loaded from an *.Xdefaults* file in the user's home directory; these resources are only available on the local machine.
4. Screen-specific resources loaded into the **SCREEN_RESOURCES** property of the root window with *xrdb*. The resource manager sorts and places the resources in **RESOURCE_MANAGER** (where they apply to all screens) or in **SCREEN_RESOURCES** (where they apply to the appropriate screen). This property is used when multiple screen types, such as monochrome and color, are differentiated.
5. The contents of any file specified by the shell environment variable `XENVIRONMENT`. If this variable has not been defined, the resource manager looks for a file named *.Xdefaults-hostname* in the user's home directory, where *hostname* is the name of the host on which the client is running. The contents of these files are used to set user- and machine-specific resources.
6. Any values specified on the command line with the *-xrm* option are loaded for that instance of the program.

The resource specifications from these various sources are loaded and merged according to the precedence rules described earlier in the section "Precedence Rules for Resource Specification". The client then merges the various defaults specified by the user with its own internal defaults, if any.

Finally, if the user has specified any command-line options (other than *-xrm*), those values override any specified by resource defaults, regardless of their source.

Testing and Editing Resources with editres

By now you have an idea of the potential complexities surrounding resource settings and how they are interpreted. The *editres* client is a resource editor that lets you examine the often-complicated hierarchy of widgets used by a client and to test different resource specifications to find the appropriate settings. For an X Toolkit-based client, *editres* lets you:

- Display and examine the client's widget hierarchy.
- Display the resources that may be set for a particular widget.
- Create resource specifications.
- Dynamically apply the new specifications to an instance of the client already running on the display.
- Write the new definitions to your own resource file.

Although it can be incredibly helpful, *editres* is not simple to use. The following sections should give you a good idea of what *editres* can do, but they don't cover every feature.

editres shows you all the resources that can be set for a widget, but it does not differentiate between those you can set at the user level and those that must be set by programming routines. Generally, the client's documentation, including its manpage, will (should!) describe the resources that you can set.

Note that *editres* only works with clients that understand the so-called *editres protocol*. Most clients built using the Athena widget set will work with *editres*. A client built using another toolkit (e.g., Qt or Gtk) may not be compatible. If you try to use *editres* with an incompatible client, the following message is displayed in the *editres* window:

```
| It appears that this client does not understand the Editres Protocol. |
```

What Widget Is That, Anyway?

Let's consider a scenario in which *editres* would be helpful to a user. Say you use *xcalc* with reverse polish notation (*rpn*), which makes it look like an HP-10C calculator. Your children, on the other hand, prefer *xcalc* to behave "normally." Their definition of "normal" means that *xcalc* should emulate a Texas Instruments TI-30 calculator. You want to change the colors of some buttons on your HP-10C calculator but don't want your changes to influence their TI-30 *xcalc*. Fortunately, *xcalc* uses differently named widgets for its buttons, depending on which calculator style you're using.

The specification:

```
| xcalc*foreground: yellow |
```

sets the foreground to yellow in every widget for which a foreground can be set. But that's not what we want. Instead, we'll use *editres* to determine the particular widgets for which to set the foreground.

In order for *editres* to examine a client's widget hierarchy, the client must be running, so we must run both *xcalc* and *editres*:

```
| xcalc -rpn &  
| editres & |
```

Figure 12-3 shows these two clients.



Figure 12-3: *xcalc* and *editres*

The *editres* Menus

If you look at the top of the *editres* window, you'll see two menus: *Commands* and *Tree*. The most important things the *Commands* menu allows you to do are:

- Display a client's widget tree (Get Widget Tree)
- Access a subwindow (called the *resource box*) from which you can test, set, and save resource specifications (Show Resource Box)
- Quit *editres*

The *Tree* menu helps you to:

- Determine the correspondence between the widget tree and the actual widgets in the client (Select Active Widget; Flash Active Widgets)
- Select groups of widgets (parents, children, etc.) for subsequent operations (e.g., showing the widget in the actual client with Flash Active Widgets)

Displaying the widget tree

To determine what resource line to use to specify the foreground for *xcalc* buttons, we must display *xcalc*'s widget tree by selecting *Get Widget Tree* from the *Commands* menu. The pointer turns into a cross and the following message appears in the message area:

| Click the mouse pointer on any toolkit client. |

Click the pointer anywhere on the *xcalc* window and the client's widget hierarchy is displayed in tree format in the application widget tree area, which is the large empty space at the bottom of the *editres* window. *xcalc* has a simple hierarchy, but even so, only part of the tree can be seen in the *editres* window at any one time, as shown in Figure 12-4.

Notice that the box beneath the *Commands* menu button is now smaller. This box is called the *panner*, and it is actually a tool that allows you to scan the entire tree. The size and location of the panner in the larger square surrounding it suggests the portion of the widget tree that is visible--in this case, approximately the top third of the tree. To view the rest of the tree, place the pointer on the panner, hold the first pointer button, and drag. The widget tree window scrolls to reveal the rest of the widget tree.

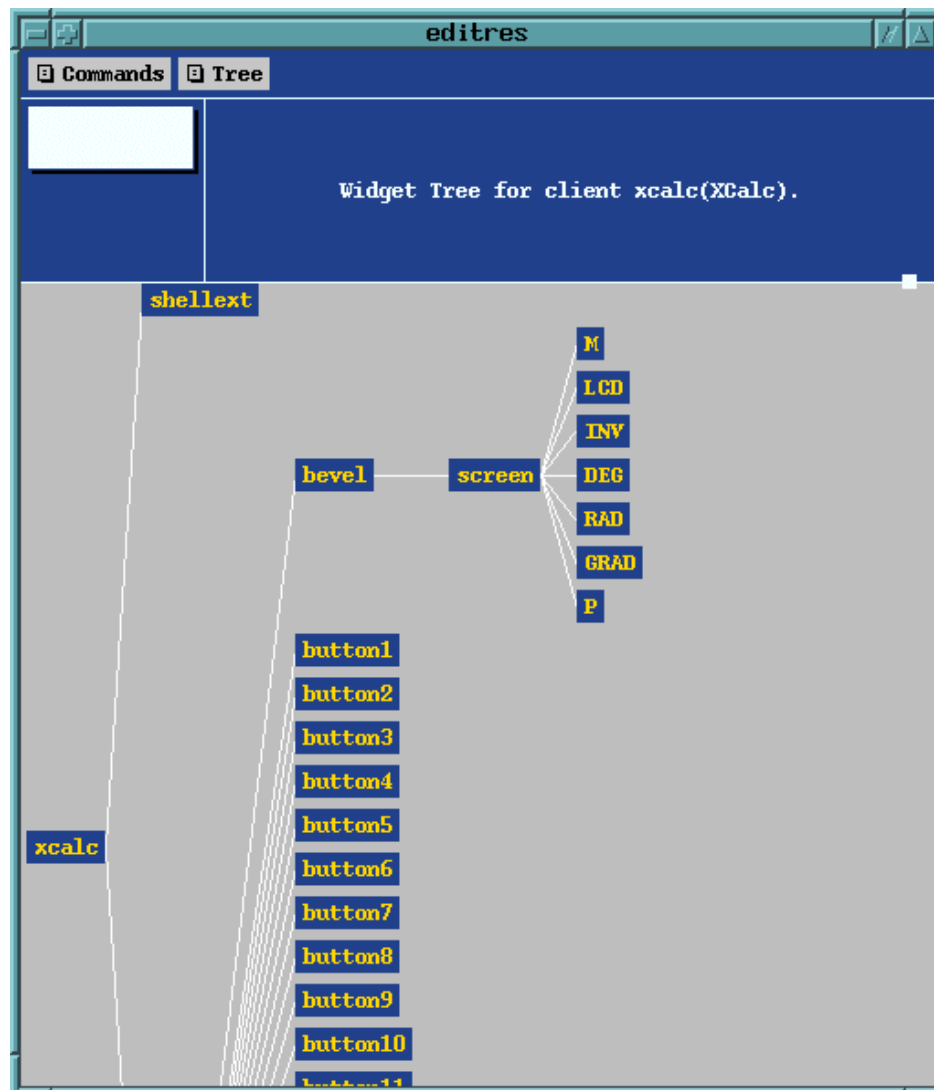


Figure 12-4: *editres* displays *xcalc*'s widget tree

Tracking down the widgets

As an example of locating widgets, assume that we wish to set the foreground color for *xcalc* buttons **0** through **9** (i.e., the digit keys). Figure 12-4 displays a few widgets such as **button1** that are likely candidates. We can determine where the **button1** widget appears in the *xcalc* application by doing the following:

1. Place the pointer on the **button1** square in the *editres* tree and click the first pointer button to select and highlight the widget (certain menu actions affect only highlighted widgets).
2. Select *Flash Active Widgets* from the *Tree* menu. The highlighted widget “flashes” in the *xcalc* window.

When we do this, we see that **button1** is the first button (the square root ($\sqrt{}$) symbol), not the button for the digit 1. Deselect the **button1** widget by clicking on it again. This time select **button7**. Reselect *Flash Active Widgets* from the *Tree* menu. Good, this time the digit 7 flashed. Note, though, that it’s pure coincidence that the seventh button also represents the digit 7.

Using the resource box to create a specification

Now that we have one of the buttons we want, we can write a resource to specify the color. To do that, select the **button7** widget in the tree if you deselected it. Then select *Show Resource Box* from the *Commands* menu. A new window, the resource box, appears on top of the main *editres* application window. The resource box is shown in Figure 12-5.

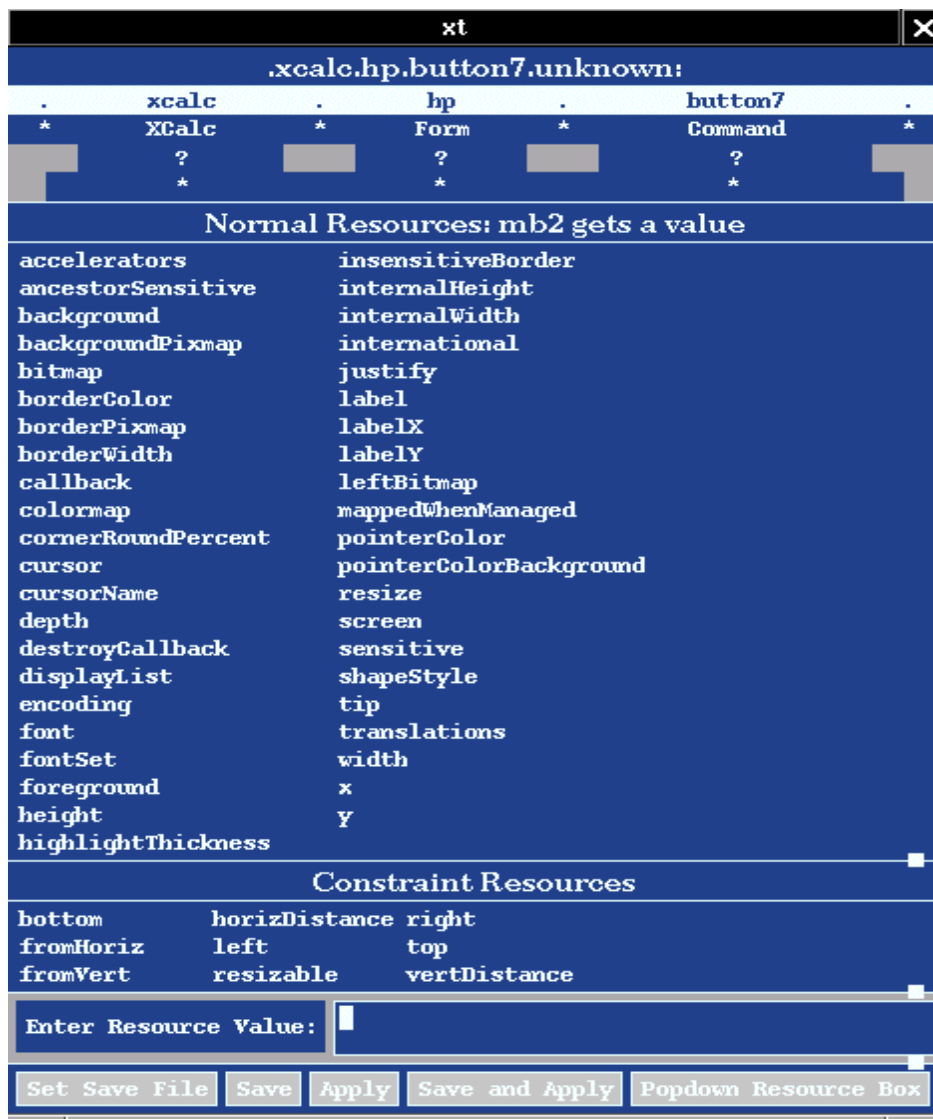


Figure 12-5: The editres resource box for Button7

Across the top of the resource box is a template resource specification for the selected widget (`.xcalc.hp.button7.unknown`), which at this stage shows the tightly bound instance name ending with an `unknown` resource variable. You can select the resource you want from the list in the box. The resource you want is `foreground`. When you click on `foreground`, the `unknown` variable in the template changes to `foreground`.

You still have to supply a value (a valid color), but first let's look at the specification more closely:

```
| .xcalc.hp.button7.foreground:
```

editres provides a way for you to edit the template resource specification. Notice that below the template are four lines of text, the first of which matches the full instance name in the template, with each component (including connectors) highlighted. The next line down shows the full class name with loose bindings. (We'll discuss the third and fourth lines in the next section.) As you can see, the four lines are spaced so that the components and connectors fall into columns. These lines provide four sets of alternatives for each of the components in the template resource specification. As you move the pointer around among the various choices, notice that a box highlights each one in turn. You can change any part of the template specification by clicking on an alternative in the same column.

For instance, to switch any tight binding in the template to a loose binding, simply click on the corresponding loose binding in the class-name line. The highlighting for that column switches to loose binding on the class-name line and the template is redrawn to include the asterisk.

Now you must enter a value for the *button7* foreground color resource in the text window near the lower-right corner of the *editres* window. The phrase "Enter Resource Value:" appears to the left of the text window. To enter a value, place the pointer in the text window and simply type "yellow".

You can test your specification on the currently running *xcalc* client. Just click on *Apply*. If the template resource can be applied successfully to the client in question, the message area to the right of the panner displays:

```
| SetValues was Successful.
```

You can verify that the new foreground resource has been incorporated into the running client by observing that the digit 7 button (*button7*) is now yellow.

The command buttons at the bottom of the resource box help you either apply the custom resource specification to the running client or save the specification in a resource file. The Set Save File button prompts you to specify the resource filename:

```
| Enter file to dump resources into:
```

If you don't select this button before trying to save your resource specification, you'll also see this prompt. One way or another, you have to tell *editres* where to save the information—generally to your resources file.

Some experimenting with *editres* shows that the widget buttons 8, 9, 17, 18, 19, 27, 28, 29, and 36 represent the other nine calculator digit buttons in the order 8, 9, 4, 5, 6, 1, 2, 3, and 0. This means that the following nine additional resources are needed to achieve the ten yellow digit buttons:

```
| .xcalc.hp.button7.foreground: yellow
|xcalc.hp.button8.foreground: yellow
|xcalc.hp.button9.foreground: yellow
|xcalc.hp.button17.foreground: yellow
|xcalc.hp.button18.foreground: yellow
|xcalc.hp.button19.foreground: yellow
|xcalc.hp.button27.foreground: yellow
|xcalc.hp.button28.foreground: yellow
|xcalc.hp.button29.foreground: yellow
|xcalc.hp.button36.foreground: yellow
```

Note that the order in which we specify the buttons has no influence on the final result.

Other ways to specify the same resource

Our sample resource line:

```
| .xcalc.hp.button7.foreground: yellow
```

only influences button 7. We added an additional nine resources for the other nine digit buttons. The following resources influence all *xcalc*'s buttons when using its *hp* mode:

```
| .xcalc.hp.?.foreground: yellow  
| .xcalc.hp*foreground: yellow
```

These resource specifications illustrate the use of the alternative components in the third and fourth lines below the template in the resource box depicted in Figure 12-5.

Selecting *Any Widget* replaces the component with the question-mark wildcard, which represents exactly one widget level. (See “Using ? as a Wildcard” earlier in this chapter.) If you go back to our example in the last section and select *Any Widget* in the third column, the question mark replaces the menu widget in the template:

```
| .xcalc.hp.?.foreground: yellow
```

If you apply this resource specification, you'll find it accomplishes the same thing our complete specification does.

Selecting *Any Widget Chain* removes the component from the template line and replaces the surrounding components using an asterisk wildcard (loose binding). (See “Tight and Loose Bindings” earlier in this chapter.) Again, go back to our example in the last section and this time select *Any Widget Chain* in the third column. An asterisk replaces the menu widget in the template and the surrounding tight bindings are removed.

```
| .xcalc.hp*foreground: yellow
```

Again, this resource accomplishes the same thing.

Of course, there are several other resource specifications you could use, but the most important thing to remember is that *editres* lets you experiment.