CHAPTER 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 *.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.

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 UNIX applications rely on command line options to allow users to customize the way they work. As we've already discussed in Chapter 11, *Command Line Options*, X applications support command line options too, but often not for all features. Also, there can be so many customizable features in an application that entering a command line to set them all would be completely impractical. (Imagine the aggravation of misspelling an option in a command that was three lines long!)

X offers an alternative to customizing an application on the command line. Almost every feature of a program can be controlled by a variable called a *resource*; you can change the behavior or appearance of a program by changing the *value* associated with a resource variable. (All of the standard X Toolkit *Command Line Options* described in Chapter 11 have corresponding resource variable names. See Table 11-1 and Table 11-2 for more information.)

Resource variables may be Boolean (such as `scrollBar: True`) or take a numeric or string value (`borderWidth: 2` or `foreground: blue`). What's more, in applications written with the X Toolkit (or an Xt-based toolkit such as the Motif toolkit), resources may be associated with separate *objects* (or "widgets") within an application. There is a syntax that allows for separate control over both a *class* of objects in the application and an individual *instance* of an object. This is illustrated by these resource specifications for a hypothetical application called *xclient*:

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

The first resource specification makes the foreground color of all buttons in the *xclient* application (in the class `Buttons`) blue; the second resource specification makes the foreground color of the `help` button in this application (an instance of the class `Buttons`) red.

The values of resources can be set as application defaults using a number of different mechanisms, including resource files in your home directory and a program called *xrdb* (X resource database manager). As we'll see, the *xrdb* program stores resources directly in the server, making them available to all clients, regardless of the machine the clients are running on.

Placing resources in files allows you to set many resources at once, without the restrictions encountered when using command line options. In addition to a primary resource file (often called *.Xdefaults* or *.Xresources*) in your home directory, which determines defaults for the clients you yourself run, the system administrator can create system-wide resource files to set defaults for all instances of a given application run on this machine. It is also possible to create resource files to set some resources only for the local machine, some for all machines in a network, and some for one or more specific machines.

The various resource files are automatically read in and processed in a certain order within an application by a set of routines called the *resource manager*. The syntax for resource specifications and the rules of precedence by which the resource manager processes them are intended to give you the maximum flexibility in setting resources with the minimum amount of text. You can specify a resource that controls only one feature of a single application, such as the red `help` button in the hypothetical *xclient* settings above. You can also specify a resource that controls one feature of multiple objects within multiple applications with a single line.

It is important to note that command line options normally take precedence over any prior resource settings; so you can set up the files to control the way you *normally* want your application to work and then use command line options to specify changes you need for only one or two instances of the application.

In this chapter, we'll first look at the syntax of resource specifications. Then we'll consider some methods of setting resources, primarily some special command line options and the *xrdb* program. Finally, we'll take a brief look at other sources of resource definition, additional files that can be created or edited to set application resources.

12.1 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. The variables for each client are documented on its reference page in Part Three of this guide.

Most of the common clients are written to use the X Toolkit. As described in Chapter 8, *Other Standard Clients*, 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, scrollbars, and so on. As we'll see, the naming syntax for certain resources parallels the object hierarchy that is built into X Toolkit programs.†

In addition, X Toolkit clients recognize a set of Core resource variables, listed in Table G-1. However, though all Toolkit applications recognize these variables, not all applications make use of them. This fine distinction is addressed in Appendix I, *Athena Widget Resources*, which gives a more technical discussion of how widgets use resources, and how applications use widgets. Appendix G also gives a detailed listing of the resources defined by each of the Athena widgets.

The most basic line 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. The following line specifies that all instances of the *xterm* application have a scrollbar:

```
xterm*scrollBar: True
```

If the name of the client is omitted, the variable applies to all instances of all clients (in this case, all clients that can have a scrollbar). If the same variable is specified as a global variable and a client-specific variable, the value of the client-specific variable takes precedence for that client. Note, however, that if the name of the client is omitted, the line should generally begin with an asterisk.

Be sure not to inadvertently omit the colon at the end of a resource specification. This is an easy mistake to make and the resource manager provides no error messages. If there is an error in a resource specification (including a syntax error such as the omission of the colon or a misspelling), the specification is ignored. The value you set will simply not take effect. To include a comment in a resource file or comment out one of the resource specifications, begin the line in question with an exclamation point (!). If the last character on a line is a backslash (\), the resource definition on that line is assumed to continue on the next line.

12.1.1 Syntax of Toolkit Client Resources

As mentioned above, X Toolkit applications (and Xt-based toolkit applications) are made up of predefined components called widgets. There can be widgets within widgets (e.g., a command button within a dialog box). The syntax of resource specifications for Toolkit clients parallels the levels of the widget hierarchy. Accordingly, you should think of a resource specification as having this format:

object.subobject[.subobject... .attribute : value

where:

† If a client was built with the X Toolkit, this should be noted on the reference page. In addition to certain application-specific resource variables, most clients that use the X Toolkit recognize a common set of resource variables, listed in Table 12-1.

object is the client program or a specific instance of the program. (See "The –name Option" later in this chapter.)

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

attribute is a feature of the last ***subobject*** (perhaps a command button), such as background color or a label that appears on it.

value is the actual setting of 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 or from the description of the resource variable on the reference page. Most of these values are similar to those used with the command line options described in Chapter 9.

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. Logical values, such as the values taken by `scrollBar`, can generally be specified as: `on` or `off`; `yes` or `no`; or `True` or `False`.

12.1.2 Tight Bindings 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:

- By a *tight* binding, represented by a dot (.).
- By a *loose* binding, represented by an asterisk (*).

A tight binding means that the components on either side of the dot must be next to one another in the widget hierarchy. A loose binding is signaled by an asterisk, a wildcard character which means there can be any number of levels in the hierarchy between the two surrounding components.

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, this resource specification to request that *xterm* windows be created with a scrollbar doesn't work:

```
xterm.scrollBar:True
```

The previous specification ignores the widget hierarchy of *xterm*, in which the VT102 window is considered to be one widget, the Tektronix window another, and the menus a third. This means that if you want to use tight bindings to request that *xterm* windows be created with a scrollbar, you should specify:

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

Of course rather than decipher the widget hierarchy (which may even change with subsequent versions of an application), it is far simpler just to use the asterisk connector in the first place:

```
xterm*scrollBar:True
```

In an application that supports multiple levels of widgets, you can mix asterisks and periods. In general, though, the developers of X recommend always using the asterisk rather than the dot as the connector even with simple applications, since this gives application developers the freedom to insert new levels in the hierarchy as they produce new releases of an application.

12.1.3 Instances and Classes

Each component of a resource specification has an associated *class*. Several different widgets, or widget attributes, may have the same class. For example, in the case of *xterm*, the color of text (`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 wanted to make the text, the pointer, and the cursor dark blue, you could specify either:

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

or:

```
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.

The real power of class and instance naming is not apparent in applications such as *xterm* that have a simple widget hierarchy. In complex applications written with the X Toolkit or the Motif Toolkit, class and instance naming allows you to do such things as specify that all buttons in dialog box be blue but that one particular button be red. For example, in the hypothetical *xclient* application, you might have a resource file that reads:

```
xclient*buttonbox*Buttons*foreground:blue
xclient*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. Instance names can be used to specify exceptions to the rules outlined by the class names. Note that a class name can be used with a loose binding to specify a resource for all clients. For example, this specification would say that the foreground colors for all clients should be blue:

```
*Foreground:blue
```

The reference page 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 is simply the primary or most obvious instance of the class.

12.1.4 Precedence Rules for Resource Specification

Even within a single resource file, such as *.Xresources*, resource specifications often conflict. For instance, recall the example from the first page of the chapter involving the hypothetical *xclient* application:

```
xclient*Buttons.foreground:blue
xclient*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 (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, which are observable in the way the resource manager interprets definitions in a user-created resource file. (The first rule applies in the previous *xclient* example.)

- 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 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 will be ignored. For example, if the program expects either:
   xterm.vt100.scrollBar:***valueinstance name*** `or:`
   XTerm.VT100.ScrollBar:***valueclass name***

   then the resource specification:
   `xterm.scrollBar: True`
   won't 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. The specification would work if you used a loose binding, however:
   `xterm*scrollBar: True`
   `(Note that the class name of` *`xterm`* `is XTerm`, not `Xterm` as you might expect.)

2. Tight bindings take precedence over loose bindings. That is, entries with instance or class names prefixed by a dot are more specific than entries with names prefixed by an asterisk, and more specific entries take precedence. For example,

the entry `xterm.vt100.geometry` will take precedence over the entry `xterm*geometry`.

3. Similarly, instances take precedence over classes. For example, the entry `*scrollBar` will take precedence over the entry `*Scrollbar`.

4. An instance or class name that is explicitly stated takes precedence over one that is omitted. For example, the entry `xterm*scrollbar` is more specific than the entry `*scrollBar`.

5. Left components carry more weight than right components. For example, the entry `xterm*background` will take precedence over `*background`.

To illustrate these rules, let's consider the following resource specifications for the hypothetical Toolkit application *xclient*, shown in Example 12-1.

```
xclient.toc*Command.activeForeground: black
*Command.Foreground: green
```

Example 12-1. Sample resources

The program would try to match these specifications against these complete tightly bound instance and class specifications:

```
xclient.toc.messageFunctions.include.activeForegroundinstance name
Xclient.Box.SubBox.Command.Foregroundclass name
```

Note that these specifications are the instance and class names for the same resource. 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.

Both resource specifications in Example 12-1 match these instance and class names. However, with its tight bindings and instance names,
`xclient.toc*Command.active%Foreground`
matches more explicitly (i.e., with higher precedence). The resource is set: the `foreground` color of the `include` button in its `active` state is set to `black`.

The specification `*Command.Foreground` also matches the instance and class names but is composed entirely of class names which are less specific; thus, it takes lower precedence than the first line in Example 12-1 (which sets the `include` button to `black`).

However, since the second line is an acceptable specification, hypothetically it would set the foreground color of other objects in the `Command` class. This resource would be set for *xclient*, as well as any other application, since the line begins with the asterisk wildcard. So if there were other *xclient* command buttons comparable to the `include` button in the hierarchy, this second line would set the foreground color of these buttons to `green`. If you want a more detailed description of how resource precedence works, see Section 9.2.3 of Volume Four, *X Toolkit Intrinsics Programming Manual*.

12.1.5 Some Common Resources

Each Toolkit command line option (listed in Table 11-1, ". MIT Toolkit and OLIT Standard Options," on page 283) has a corresponding resource variable. Most X Toolkit (and OLIT Toolkit) applications recognize some subset of these resources.

Table 12-1 lists the resource variables recognized by most Toolkit clients.

Table 12-1. Common Toolkit Resources

Instance Name	Class Name	Default	Description _
background	`Background`	White	Background color.
foreground	`Foreground`	Black	Foreground color.
borderColor	`BorderColor`	Black	Border color.
borderWidth	`BorderWidth`	1 pixel	Border width.

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

```
xclient.background xclient.buttonBox.background
xclient.buttonBox.commandButton.background
xclient.buttonBox.quit.background
```

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

Of course, the specification:

```
xclient*background
```

would match any and all of them.

Appendices list resources for the OLIT and Athena widgets.

12.2 Event Translations

We've discussed the basics of resource naming syntax. From the sample resource settings, it appears that what many resource variables do is self-evident or nearly so. Among the less obvious resource variables, there is one type of specification, an event translation, that can be used with many clients and warrants somewhat closer examination.

User input and several other types of information pass from the server to a client 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, *An Introduction to OPEN LOOK and the X Window System*, moving the pointer or pressing a key, etc., 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*.

Many event translations are programmed into an application and are invisible to the user.* For our purposes we are only concerned with very visible translations of certain input events, primarily the translation of keystrokes and pointer button clicks to particular actions by a client program.†

12.2.1 The Syntax of Event Translations

The operation of many clients, notably *xterm*, is partly determined by default input event translations. For example, as explained in Appendix A, *The xterm/olterm Terminal Emulator*, selecting text with the first pointer button (an event) saves that text into memory (an action).

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

1. Pressing the first pointer button.
2. Moving the pointer while holding down the first button.
3. Releasing the button.

Each of these input events performs a part of the action of selecting text:

1. Unselects any previously selected text and begins selecting new text.
2. Extends the selection.
3. Ends the selection, saving the text into memory (both as the PRIMARY selection and CUT_BUFFER0).

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

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

where 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 should be terminated by a newline character (\n), which is in turn followed by a backslash (\) to escape the actual newline.

These are default translations for *xterm*.‡

† For more information on events and translations, see Volume Four, *X Toolkit Intrinsics Programming Manual*.

‡ They are actually slightly simplified versions of default translations. Before you can understand the actual translations listed on the *xterm* reference page in Part Three of this guide, you must learn more about the syntax of translations. In addition to the current chapter, read Appendix F, *X Toolkit Translation Table Syntax*.

All of the events are simple, comprised of a single button motion. As we'll see, events can also have modifiers: i.e., additional button motions or keystrokes (often Control or Meta) 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.)

As you can see, the default actions listed in the table are hardly intuitive. The event-action mappings that can be modified using translation resources are usually described on the reference page for the particular client.

You can specify non-default translations using a translation table (a resource containing a list of translations). Since actions are part of the client application and cannot be modified, what you are actually doing is specifying alternative events to perform an action.* Keep in mind that only applications written with the X Toolkit (or an Xt-based toolkit such as the Motif 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 is basically like any other resource specification with a few exceptions. First, the final ***argument*** is always `translations`, indicating that one (or more) of the event-action bindings associated with the `[`***object***`*[`***subobject...***`]]` are being modified.†

Second, note that `#override` is not the ***value*** of the resource; it is literal and indicates that what follows should override any default translations. In effect, `#override` is no more than a pointer to the true ***value*** of the resource: a new event-action mapping (on the following line), where the event may take a modifier.

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 match the event(s) of a default translation *exactly*. If the new translation does not conflict with any existing translation, it is merely appended to the defaults.‡

In order to be specified as a resource, a translation table must be a single string. The `#override` is followed by a backslash (\) to indicate that the subsequent line should be a continuation of the first.

In the previous basic syntax example, the ***value*** is a single event-action mapping. The ***value*** could also be a list of several mappings, linked by the characters "\n\" to make the resource a continuous string.

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

† As we'll see, in certain cases you may be able to supply an alternative *argument* (such as a selection name) to an action. These changes *are* interpreted by the resource manager.

‡ The use of modifiers can actually become quite complicated, sometimes involving multiple modifiers. For our purposes, we'll deal only with simple modifiers. For more information on modifiers, see Appendix F in this guide and Volume Four, *X Toolkit Intrinsics Programming Manual*.

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

12.2.1.1 xterm Translations to Use xclipboard

As explained in Chapter 5, 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. See the discussion of *xclipboard* in Chapter 8, *Other Standard Clients* before proceeding.

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. If you are running *xclipboard* and you copy text to be made the CLIPBOARD selection, this text automatically appears in the *xclipboard* window.

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

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

According to this translation table, while selecting text with `Button1` (the modifier), the event of pressing the third pointer button (`Btn3Down`), while continuing to hold down the first button, produces the action of making the text the CLIPBOARD selection. (Notice that we've taken the `select-end` action and combined it with the argument `CLIPBOARD`. The default translation uses the arguments `PRIMARY,CUT_BUFFER0`.)

The second line modifies the way selected text is pasted into a window so that the `CLIPBOARD` selection can be pasted. As described in Chapter 5, pressing the second pointer button pasted the contents of the PRIMARY selection, by default. 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 releasing pointer button 2, while pressing any modifier button or key other than Control or Meta, inserts text from the PRIMARY selection or, if the selection is empty, from cut buffer 0. In the second line of our translation table, we've replaced `CUT_BUFFER0` with the `CLIPBOARD` selection. The new behavior is that releasing the second pointer button pastes the PRIMARY selection, or if there is none, the CLIPBOARD selection.

Thus, according to the translations in the example, if you select text as usual with the first pointer button, and then additionally 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, as shown in Figure 12-1.

Figure 12-1. Selected text appears automatically in the xclipboard window

Since our first translation specifies a different event/action mapping than the default translation for selecting text (discussed in the previous section), the default translation still applies. If you select text with the first pointer button alone, that text is still made the PRI-

MARY selection and fills CUT_BUFFER0. To send text to the *xclipboard*, you would need to press the third pointer button as well; thus, not all selected text needs to be made the CLIPBOARD selection (and sent automatically to the *xclipboard*).

There are advantages to making only certain selections CLIPBOARD selections. You can keep *xclipboard* running and make many text selections by the default method (first pointer button), without filling up the *xclipboard* window. And chances are you don't want to save every piece of text you copy for an extended period of time, anyway.

The CLIPBOARD selection and the *xclipboard* client also get around the potential problems of selection ownership discussed in Chapter 5. Once text becomes the CLIPBOARD selection, it is owned by the *xclipboard* client. Thus, if the client from which text was copied (the original owner) goes away, the selection is still available, owned by the *xclipboard*, and can be transferred to another window (and translated to another format if necessary).

12.2.1.2 Entering Frequently Used Commands with Function Keys

The sample *xterm* translations to use the *xclipboard* client involve just a few of the actions *xterm* recognizes. Among the more useful translations you can specify for *xterm* are function key mappings that allow 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 such a function key mapping is fairly simple. The following line maps the text string "lpq -Pprinter1" (the BSD 4.3 command to check the queue for the printer named printer1) to the F1 function key:

```
<Key>F1: string("lpq -Pprinter1")
```

Notice the quotes surrounding the text string. If the argument to `string` includes spaces or non-alphanumeric characters, the whole argument must be enclosed in one pair of double quotes. (Don't make the mistake of quoting individual words.)

The translation table would be:

```
*VT100.Translations:#override\
    <Key>F1: string("lpq -Pprinter1")
```

This sample translation causes `lpq -Pprinter1` to be passed to the command line in the active *xterm* window when you press the F1 function key, as in Figure 12-2.

Notice, however, that the command is not invoked because there has been no carriage return. The sample translation does not specify a return. You can add a return as the argument to another `string` action within the same translation.

To specify the Return (or any) key, use the hexadecimal code for that key as the argument to `string`. Keycodes and the procedure for determining them are explained in Chapter 14, *Customization Clients*. The letters "0x" signal a hexadecimal key code. If you want to enter a key as an argument to `string`, use "0x" followed by the specific code. The code for the Return key is "d" or "0d."†

† As explained in Chapter 14, *Customization Clients*, the command `xmodmap -pk` returns a long listing of all keycodes. The codes have the either of the following forms:

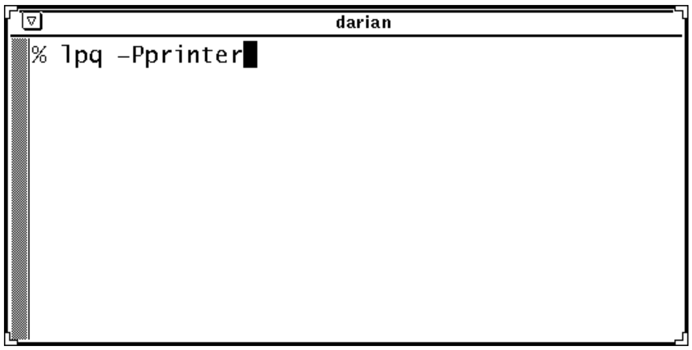

Figure 12-2. Pushing F1 passes command text to xterm shell .

```
0xffab
0x00ab
```

where *ab* represents two alphanumeric characters. To specify a key as an argument to `string`, you can omit the "ff" or "00" in the *xmodmap* listing.

The following translation table specifies that pressing F1 passes the line `lpq -Pprinter1` followed by a carriage return to an *xterm* window:

```
*VT100.Translations:#override\
<Key>F1: string("lpq -Pprinter1") string(0x0d)
```

Remember, you can list several translations in a single table. The following table maps function keys F1 through F3:

```
*VT100.Translations:#override\
    <Key>F1: string("lpq -Pprinter1") string(0x0d)\n\
    <Key>F2: string("cd ~/bitmap;ls") string(0x0d)\n\
    <Key>F3: string("cd /usr/lib/X11") string(0x0d)
```

According to these translations, pressing F2 inserts the command string

```
cd ~/bitmap;ls
```

which changes directory to *~/bitmap* and then lists the contents of that directory. Notice that you can issue multiple commands (*cd* , *ls*) with a single key. Pressing F3 changes directory to */usr/lib/X11*.

Keep in mind that all the translations for an application can appear in the same table. For example, we can combine the *xterm* translations to use the *xclipboard* with the translations to map function keys.

```
*VT100.Translations:#override\
Button1 <Btn3Down>: select-end(CLIPBOARD)\n\
~Ctrl ~Meta <Btn2Up>: insert-selection(PRIMARY,CLIPBOARD)\n\
<Key>F1: string("lpq -Pprinter1") string(0x0d)\n\
<Key>F2: string("cd ~/bitmap;ls") string(0x0d)\n\
<Key>F3: string("cd /usr/lib/X11") string(0x0d)
```

The order of the translations is not important. However, it is necessary to end all but the final line with the sequence "\n\" to make the resource a continuous string.

12.2.1.3 Other Clients that Recognize Translations

xterm is not the only client whose operation can be modified by specifying event translations as resources (though it is probably the client you'll be most interested in modifying). Among the standard clients, *xbiff*, *xcalc*, *xdm*, *xman*, and *xmh* all recognize certain actions that can be mapped to particular keys or key combinations using the translation mechanism. See the relevant client reference pages in Part Three of this guide for complete lists of actions.

You can also modify the operation of the Text widget used by *xedit*, *xmh*, and other X Toolkit applications. See Appendix I, *Athena Widget Resources*, for a list of actions recognized by the Text widget. Keep in mind, however, that the default Text widget recognizes dozens of commands, which are summarized in the discussion of *xedit* in Chapter 8, *Other Standard Clients*. It may not be practical or desirable to modify them all.

If you choose to modify the Text widget, you can do so for all relevant clients by introducing the translations with the line:

```
*Text*Translations: #override\
```

You can also specify different translations for different clients that use the widget by prepending the client's name. To affect the operation of the Text widget only under *xedit*, introduce the translation table with the line:

```
Xedit*Text*Translations: #override\
```

In modifying the operation of the Text widget, keep in mind that insert mode is the default. In other words, like *emacs*, most of the individual keystrokes you type are added to the text file; an exception is Backspace, which predictably deletes the preceding character. The commands to move around in a file, copy and delete text, etc., involve a combination of keys, one of which is generally a modifier key. If you want to modify a command, you should use an alternative key combination, rather than a single key.

For example, the following table offers two suitable translations:

```
*Text*Translations: #override\
    Meta<Key>f:next-page()\n\
    Meta<Key>b:previous-page()
```

The first translation specifies that pressing the key combination Meta-f moves the cursor ahead one page in the file (scrolls the file forward one window); the second translation specifies that Meta-b moves the cursor back one page. The actions performed are fairly obvious from their names. For a complete list of actions recognized by the Text widget, see Appendix G.

For more information about events, actions, and translation table syntax, see Appendix F, *X Toolkit Translation Table Syntax*, and Volume Four, *X Toolkit Intrinsics Programming Manual*.

12.3 How to Set Resources

Learning to write resource specifications is a fairly 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 our purposes, we are primarily concerned with specifying resources for a single user running applications both on the local system and on remote systems in a network.

As we've said, resources are generally specified in files. A resource file can have any name you like. Resources are generally "loaded" into the X server by the *xrdb* client, which is normally run from your startup file or run automatically by *xdm* when you log in. (See , for information about startup files and *xdm*.) Prior to Release 2 of X, there was only one resource file called *.Xdefaults*, placed in the user's home directory. If no resource file is loaded into the server by *xrdb*, the *.Xdefaults* file will still be read.

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 was that users who were running clients on multiple machines had to 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 Definition" later in this chapter for information on setting machine-specific resources. This section gives an overview of additional ways to specify resources using a variety of system files.

In addition to loading resource files, you can specify defaults for a particular instance of an application from the command line using two options: `-xrm` and `-name`.

First we'll consider a sample resources file. Then we'll take a look at the use of the `-xrm` and `-name` command line options. Finally, we'll discuss various ways you can load resources using the *xrdb* program and consider other sources of resource definition, later in this chapter.

12.3.1 A Sample Resources File

Figure 12-3 shows a sample resources file. This file sets the border width for all clients to a default value of two pixels, and sets other specific variables for *xclock* and *xterm*. The meaning of each variable is obvious from its name (for example, `xterm*scrollBar: True` means that *xterm* windows should be created with a scrollbar.

Note that comments are preceded by an exclamation point (!).

For a detailed description of each possible variable, see the appropriate client reference pages in Part Three of this guide.

```
*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
```

Figure 12-3. A sample resources file

12.3.2 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.

12.3.2.1 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' &
```

Note that a resource specification on the command line must be quoted using the single quotes in the line above.

The `-xrm` option only specifies the resource(s) 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 classes, since most clients have command line options that correspond to instance variable names. For example, the `-fg` command line option sets the `foreground` attribute of a window, but `-xrm` must be used to set `Foreground`.

Note also that a resource specified with the `-xrm` option will 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
```

This command line specification of another cursor will fail:

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

because the resource `xterm*pointerShape` is more specific than the resource `*pointerShape`. Instead, you'll get an *xterm* with the previously specified pirate cursor.

To override the resource database (and get the Gumby cursor), you'd need to use a resource equally (or more) specific, such as the following:

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

12.3.2.2 The –name Option

The `-name` option lets you name one instance of an application; the server identifies the single instance of the application by this name. The name of an application affects how resources are interpreted.

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

```
% xterm -name bigxterm &
```

When this command is run, the client uses any resources specified for `bigxterm` rather than for `xterm`.

The `-name` option allows you to create different instances of the same application, each using different resources. For example, you could put the following entries into a resource file such as *.Xresources*:

```
XTerm*Font: 8x13
smallxterm*Font: 6x10
smallxterm*Geometry: 80x10
bigxterm*Font: 9x15
bigxterm*Geometry: 80x55
```

You could then use these commands to create *xterm*s of different specifications:

```
% xterm &
```

would create an *xterm* with the default specifications, while:

```
% xterm -name bigxterm &
```

would create a big *xterm*, 80 characters across by 55 lines down, displaying in the font 9x15. The command:

```
% xterm -name smallxterm &
```

would create a small *xterm*, 80 characters across by 10 lines down, displaying in the font 6x10.

12.4 Setting Resources with xrdb

The *xrdb* program saves you from the difficulty of maintaining multiple resource files if you run clients on multiple machines. It stores resources in the X server, where they are accessible to all clients using that server. (Technically speaking, the values of variables are stored in a data structure referred to as the RESOURCE_MANAGER property of the root window of screen 0 for that server. From time to time, we may refer to this property colloquially simply as the resource database.)

The appropriate *xrdb* command line should normally be placed in your *.xinitrc* file or *.xsession* file to initialize resources at login, although it can also be invoked interactively. It has the following syntax:

xrdb [***options*** [***filename***

The *xrdb* client takes several options, all of which are documented on the reference page in Part Three of this guide. Several of the most useful options are discussed in subsequent sections. (Those that are not discussed here have to do with *xrdb*'s ability to interpret C preprocessor-style defined symbols; this is an advanced topic. For more information, see the *xrdb* reference page in Part Three of this guide, and the *cpp* (1) reference page in your *UNIX Reference Manual*.)

The optional *filename* argument specifies the name of a file from which the values of client variables (resources) will be read. If no filename is specified, *xrdb* will expect to read its data from standard input. That is, the program will appear to hang, until you type some data, followed by an end-of-file (Control-D on many UNIX systems). Note that whatever you type will override the previous contents of the RESOURCE_MANAGER property, so if you inadvertently type *xrdb* without a filename argument, and then quit with Control-D, you will delete any previous values. (You can append new settings to current ones using the `-merge` option discussed later in this chapter.)

The resource ***`filename`*** can be anything you want. Two commonly used names are *.Xdefaults* and *.Xresources*. The former is an older form, but is more commonly used in the OpenWindows environment. The latter is newer, and is more prevalent in X11R5 environments. Either form can be used.

You should load a resource file with the *xrdb* `-load` option. For example, to load the contents of your *.Xresources* file into the RESOURCE_MANAGER property, you would type:

```
% xrdb -load .Xresources
```

12.4.1 Querying the Resource Database

You can find out what options are currently set by using the `-query` option. For example:

```
% xrdb -query
XTerm*ScrollBar: True
bigxterm*font: 9x15
bigxterm*Geometry: 80x55
smallxterm*Font: 6x10
smallxterm*Geometry: 80x10
xterm*borderWidth: 3
```

If *xrdb* has not been run, this command will produce no output.

12.4.2 Loading New Values into the Resource Database

By default, *xrdb* reads its input (either a file or standard input) and stores the results into the resource database, replacing the previous values. If you simply want to merge new values with the currently active ones (perhaps by specifying a single value from standard input), you can use the `-merge` option. Only the new values will be changed; variables that were already set will be preserved rather than overwritten with empty values. In fact, if you are running with *xdm,* it is preferable to use -merge, since this will prevent you from accidentally erasing some valuable initial resources that *xdm* might have stored in the server.

For example, let's say you wanted to add new resources listed in the file *new.values*. You could say:

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

As another example, if you wanted all subsequently run *xterm* windows to have scrollbars, you could use standard input, and enter:

```
% xrdb -merge xterm*scrollBar:True
```

and then press Control-D to end the standard input. Note that because of precedence rules for resource naming, you may not automatically get what you want. For example, if you specify:

```
xterm*scrollBar:True
```

and the more specific value:

```
xterm*vt100.scrollBar:False
```

has already been set, your new, less specific setting will be ignored. The problem isn't that you used the `-merge` option incorrectly—you just got caught by the rules of precedence.

If your specifications don't seem to work, use the `-query` option to list the values in the RESOURCE_MANAGER property and look for conflicting specifications.

Note also that when you add new specifications, they won't affect any programs already running, but only programs started after the new resource specifications are in effect. (This is also true even if you overwrite the existing specifications by loading a new resource file. Only programs run after this point will reflect the new specifications.)

12.4.3 Saving Active Resource Definitions in a File

Assume that you've loaded the RESOURCE_MANAGER property from an *.Xresources* or other file. However, you've dynamically loaded a different value using the `-merge` option and you'd like to make the new value your default.

You don't need to edit the file manually (although you certainly could.) The `-edit` option allows you to write the current value of the RESOURCE_MANAGER property to a file. If the file already exists, it is overwritten with the new values. However, *xrdb* is smart enough to preserve any comments and preprocessor declarations in the file being overwritten, replacing only the resource definitions.

For example:

```
% xrdb -edit ~/.Xresources
```

will save the current contents of the RESOURCE_MANAGER property in the file *.Xresources* in your home directory.

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

```
% xrdb -edit .mydefaults -backup old
```

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

12.4.4 Removing Resource Definitions

You can delete the definition of the RESOURCE_MANAGER property from the server by calling *xrdb* with the `-remove` option.

There is no way to delete a single resource definition other than to read the current *xrdb* values into a file. For example:

```
% xrdb -query > filename
```

Use an editor to edit the file, deleting the resource definitions you no longer want and save the file:

```
% vi filename
```

Then read the edited values back into the RESOURCE_MANAGER with *xrdb*:

```
% xrdb -load filename
```

12.4.5 Listing the Current Resources for a Client: appres

The *appres* (*app*lication *res*ource) program, available as of X11 Release 4, lists the resources that currently might apply to a client. These resources may be derived from several sources, including the user's *.Xresources* file and a system-wide application defaults file. The directory */usr/lib/X11/app-defaults* contains application default files for several clients. The function of these files is discussed in the next section. For now, be aware that all of the resources contained in these files begin with the class name of the application.

Also be aware that *appres* has one serious limitation: it cannot distinguish between valid and invalid resource specifications. It lists all resources that might apply to a client, whether or not the resources are correctly specified.

appres lists the resources that apply to a client having the ***class_name*** and/or ***instance_name*** you specify. Typically, you would use *appres* before running a client program to find out what resources the client program will access.

For example, say you want to run *xterm* but you can't remember the latest resources you've specified for it, whether you've loaded them, or perhaps what some of the application defaults are, etc. You can use the *appres* client to check the current *xterm* resources. If you specify only a class name, as in this command line:

```
% appres XTerm
```

appres lists the resources that any *xterm* would load. In the case of *xterm*, this is an extensive list, encompassing all of the system-wide application defaults as well as any other defaults you have specified in a resource file.

You can additionally specify an instance name to list the resources applying to a particular instance of the client, as in:

```
% appres XTerm bigxterm
```

If you omit the class name, *xappres* assumes the class `-NoSuchClass-`, which has no defaults, and returns only the resources that would be loaded by the particular instance of the client.

Note that the instance can simply be the client name, for example, `xterm`. In that case none of the system-wide application defaults would be listed, since all begin with the class name `XTerm`. For example, the command:

```
% appres xterm
```

might return resources settings similar to these:

```
xterm.vt100.scrollBar: True
xterm*PhonyResource: youbet
xterm*pointerShape: gumby
xterm*iconGeometry: +50+50
*VT100.Translations: #override\
    Button1 <Btn3Down>: select-end(CLIPBOARD)\n\
    ~Ctrl ~Meta <Btn2Up>: insert-selection(PRIMARY,CLIPBOARD)
```

Most of these resources set obvious features of *xterm*. The translation table sets up *xterm* to use the *xclipboard*. Notice also that *appres* has returned an invalid resource called `PhonyResource` that we created for demonstration purposes. You can't rely on *appres* to tell you what resources a client will actually load because the *appres* program cannot distinguish a valid resource specification from an invalid one. Still, it can be fairly useful to jog your memory as to the defaults you've specified in your *.Xresources* file, as well as the system-wide application defaults.

12.4.6 Other Sources of Resource Definition

If *xrdb* has not been run, the RESOURCE_MANAGER property will not be set. Instead, the resource manager looks for a file called *.Xdefaults* in the user's home directory. As we discussed earlier, 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 these sources of resource definition are conIndexsulted in this order:

1. The client's application defaults file(s), if any, which usually reside in the directory */usr/lib/X11/app-defaults,* will be loaded into the resource manager (Note that the path can be reset with the XFILESEARCHPATH environment variable). Application-specific resource files generally have the name *Class*, where *Class* is the class name of the client program.

   Any other application-specific resource files: a resource file named by the variable XUSERFILESEARCHPATH; or if this variable is not set, a file in the directory named by the environment variable XAPPLRESDIR.

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.

   If no resources are loaded in this way, the resource manager looks for a *.Xdefaults* file in the user's home directory; these resources are only available on the local machine.

3. Screen-specific resources loaded into the SCREEN_RESOURCES property of the root window with *xrdb*. The resource manager will sort and place the resources in RESOURCE_MANAGER (where they will apply to all screens) or in SCREEN_RESOURCES (where they will apply to the appropriate screen).

4. Next, the contents of any file specified by the shell environment variable XENVIRONMENT will be loaded.

   If this variable is not 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 where the *client* is running (not necessarily where the display is).

   These methods are used to set machine-specific resources.

5. Any values specified on the command line with the `-xrm` option will be loaded for that instance of the program.

All of these various sources of defaults will be loaded and merged according to the precedence rules described earlier in the section "Precedence Rules for Resource Specification."

The client will then merge these various defaults specified by the user with its own internal defaults, if any.

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