Advanced Motif Window Manager Customization

This chapter describes the **mwm** resources that enable you to perform the following tasks:

- Modify menus, including the Root Menu
- Create new menus
- Modify mouse button bindings
- Modify key bindings
- Set up virtual panning for your screen
- Control access of other clients to the **mwm** menus

Understanding the Resource Description File

The supplementary **mwm** resource description file contains specifications for menus, key bindings, and mouse bindings that are referred to by entries in the `.Xdefaults` file and other files that the window manager uses to create the resource database. If you do not have a `.mwmrc` file in your home directory, copy the system resource description file into your home directory with the following command:

```
% cp /usr/lib/X11/system.mwmrc  ~/.mwmrc
```

You can freely modify the local copy of the resource description file to customize your Motif environment.

The *configFile* resource specifies the pathname for the **mwm** resource description file. If the pathname begins with ~/ (a tilde followed by a slash), **mwm** considers it to be relative to your home directory (as specified by the **HOME** environment variable). If the **LANG** environment variable is set, **mwm** looks for `$HOME/$LANG/configFile`. If that file does not exist or if **LANG** is not set, **mwm** looks for `$HOME/configFile`. If the *configFile* pathname does not begin with ~/ (a tilde followed by a slash), **mwm** considers it to be relative to the current working directory.

User actions, such as keystrokes, are associated with predefined window manager functions. Each function has a name that begins with an **f.** (f dot) character sequence. In general, the meaning of the function is evident from its name (see Table 6−1). (See the **mwm** reference page in the *Motif Programmer's Reference* for more information on these functions.)

Table 1 Window Manger Functions

Function	*Description*
f.beep	Causes a beep.
f.cci	Controls the placement and naming of client−command interface (CCI) commands generated by applications.
f.circle_down	Moves the top window to the bottom of the window stack.
f.circle_up	Moves the bottom window to the top of the window stack.
f.exec or !	Executes the following shell command.
f.focus_color	Sets the colormap focus to a window.
f.focus_key	Sets the keyboard input focus to a window.
f.goto	Moves the root window to a specified location.
f.kill	Kills an application and its window.
f.lower	Moves a window to the bottom of the window stack.
f.maximize	Maximizes a window.
f.menu	Activates the named menu. This function can be used to create cascading and Popup Menus.
f.minimize	Iconifies a window.
f.move	Starts an interactive move for a window.
f.next_cmap	Installs the next colormap.
f.next_key	Moves the keyboard input focus to the next window in the window stack.
f.nop	Does nothing.
f.normalize	Causes an icon or a maximized window to be displayed at its normal size.
f.normalize_and_raise	Causes an icon or a maximized window to be displayed at its normal size and raised to the top of the window stack.
f.pack_icons	Reorganizes the icons according to the current icon placement policy.
f.pan	Moves the root window a specified amount.
f.pass_keys	Toggles the use of special key bindings.
f.post_wmenu	Posts the Window Menu.
f.prev_cmap	This function installs the previous colormap in the list of colormaps for the window with the colormap focus.
f.prev_key	Moves the keyboard input focus to the previous window in the window stack.
f.quit_mwm	Exits the window manager without exiting the X Window System.
f.raise	Raises a window to the top of the window stack.
f.raise_lower	If obscured, raises a window to the top of the window stack; otherwise, lowers it to the bottom of the window stack.
f.refresh	Redraws all the windows on the screen.
f.refresh_win	Redraws a single window.
f.resize	Starts an interactive resize for a window.
f.restart	Stops and restarts the window manager.
f.restore	Restores an iconified window to its previous state.
f.restore_and_raise	Restores an iconified window to its previous state and raises it to the top of its stack.
f.screen	Moves a pointer to a specific screen.
f.send_msg	Sends a client message to the application.
f.separator	Draws a separator in a menu pane.
f.set_behavior	Restarts the window manager with the default behavior or reverts to any custom behavior.
f.title	Inserts a title in a menu pane.
f.track_pan	Continuously moves the root window in the direction of the mouse.

The `.mwmrc` file defines named groups of key bindings, button bindings, and menu definitions. These groups are referenced by name as values for **mwm** resources, such as *keyBindings* and *buttonBindings*. For example, if you wanted to define alternative responses for mouse button actions, you would create the named set of bindings (for instance, *MyButtonBindings*) in the `.mwmrc` file and reference those bindings in your `.Xdefaults` file as follows:

```
Mwm*buttonBindings: MyButtonBindings
```

The following sections describe the syntax for defining menus, key bindings, and button bindings.

Modifying the Window Menu

All window manager menus are defined using the following syntax:

```
Menu menu_name
{
item1 [mnemonic ] [accelerator ] function [argument ]
item2 [mnemonic ] [accelerator ] function [argument ]
item# [mnemonic ] [accelerator ] function [argument ]
}
```

The menu is given a name, and each item in the menu is given a name or graphic representation (bitmap). The item is followed by an optional mnemonic or accelerator or both, and then by one of the window manager functions listed in Table 6−1. The function is the action that the window manager takes when that menu item is selected. Some functions require an argument.

The default Window Menu definition as it appears in the `/usr/lib/X11/system.mwmrc` file is as follows:

```
Menu DefaultWindowMenu
{
Restore                 _R              Alt<Key>F5              f.normalize
Move                    _M              Alt<Key>F7              f.move
Size                    _S              Alt<Key>F8              f.resize
Minimize                _n              Alt<Key>F9              f.minimize
Maximize                _x              Alt<Key>F10             f.maximize
Lower                   _L              Alt<Key>F3              f.lower
no−label                                                        f.separator
Close                   _C              Alt<Key>F4              f.kill
}
```

Not all applications require each one of these default functions. For example, if you have a real−time application, it should never be iconified. The Motif Window Manager allows you to specify a Window Menu for each application you are using.

To modify the default Window Menu, begin by copying the *DefaultWindowMenu* definition from the `/usr/lib/X11/system.mwmrc` file to the `.mwmrc` file in your home directory. Rename the default menu definition (*MailWindowMenu,* for example) and make the appropriate changes, following the syntax shown at the beginning of this section.

Then you need to reference the alternate Window Menu definition in your `.Xdefaults` file by using the *windowMenu* resource:

```
Mwm*my_mail_program*windowMenu: MailWindowMenu
```

Remember that you need to restart **mwm** for your version of the menu to appear.

Note that the menu item names you specify in the `.mwmrc` file can be overridden by applications that rename menu item names.

Creating New Menus

To create a completely new menu, use the general menu syntax in Section 6.2 as a model and follow these steps:
1.Fill in a menu name.
2.Create the item names.
3.Choose a mnemonic and accelerator (optional).
4.Give each item a function to perform (see Table 6−1).

Menu Items

An item can be either a character string or a graphic representation (bitmap or pixmap).

A character string for items must be compatible with the menu font that is used. Character strings must be typed precisely using one of the following styles:

- Any character string containing a space must be enclosed in "" (double quotes) (for example, *"Menu name"*).
- Single−word strings do not have to be enclosed in double quotes, but it is probably a good idea for the sake of consistency (for example, *"Menuname"*).
- An alternate method of dealing with multiple−word item names is to use an underbar in place of the space (for example, *Menu_name*).

An item in either X bitmap (XBM) or X pixmap (XPM) file format can be created. Using the @ (at sign) in the menu syntax tells the window manager that what follows is the pathname for a bitmap or pixmap file:

```
@bitmapfile function [argument ]
```

The following is an example of a newly created menu. The menu is named *Graphics Projects*. The menu items are all bitmaps symbolizing different graphics projects. The bitmaps are kept in the directory `/users/pat/bits`. When the user selects a symbol, the graphics program starts and opens the appropriate graphics file.

```
Menu "Graphics Projects"
{
"Graphics Projects"                                                    f.title
@/users/pat/bits/fusel.bits                                            f.exec "ca
                                                                       /spacestar
@/users/pat/bits/lwing.bits                                            f.exec "ca
                                                                       /spacestar
@/users/pat/bits/rwing.bits                                            f.exec "ca
                                                                       /spacestar
@/users/pat/bits/nose.bits                                             f.exec "ca
                                                                       /spacestar
}
```

Another method for specifying the pathname is to replace `/users/pat/` with the ~/ (tilde and slash) characters. The ~/ specifies the user's home directory. Another method is to use the *bitmapDirectory* resource. If the *bitmapDirectory* resource is set to `/users/pat/bits`, then a menu item could be specified as follows:

```
@fusel.bits                                    f.exec "cad /spacestar/fusel.e12
```

Mnemonics and Accelerators

You can use mnemonics and keyboard accelerators in the menus that you create. Mnemonics are functional only when the menu is posted; accelerators are functional whether or not the menu is posted. A mnemonic specification has the following syntax:

```
mnemonic = _character
```

The _ (underbar) is placed under the first matching *character* in the label. If there is no matching *character* in the label, no mnemonic is registered with the window manager for that label. The accelerator specification is a key action with the same syntax as is used for binding keys to window manager functions:

```
key context function [argument ]
```

When choosing accelerators, be careful not to use key actions that are already used in key bindings. (See Section 6.5 for information about keyboard bindings.)

The following line from the default Window Menu illustrates mnemonic and accelerator syntax:

```
Restore _R Alt<Key>F5 f.normalize
```

Functions

The predefined Motif Window Manager functions are listed in Table 6−1. Each **mwm** function operates in one or more of the following contexts:

root

Operates the function when the workspace or root window is selected.

window

Operates the function when a client window is selected. All subparts of a window are considered as windows for function contexts. Note that some functions operate only when the window is in its normalized or iconified state (**f.maximize**), or its maximized or iconified state (**f.normalize**).

icon

Operates the function when an icon is selected.

Each function is activated by one or more of the following devices:

- Mouse button
- Keyboard key
- Menu item

Any selection that uses an invalid context, an invalid function, or a function that does not apply to the current context is grayed out. For example, the *Restore* selection on a terminal window's Window Menu and the *Minimize* selection on an icon's menu are invalid. Also, menu items are grayed out if they are assigned the **f.nop** (no operation performed) function.

If you want your new menu to appear whenever a certain mouse button or keyboard key is pressed, follow these steps:

1. Choose the mouse button or keyboard key that you want to use.
2. Choose the action on the button or key that causes the menu to appear.
3. Choose the context in which the menu is to appear.
4. Use the **f.menu** function with the new menu's name as an argument to bind the menu to the button or key.

For example, you may want to create a root menu that gives some control over the entire screen area. The definition of a root menu might look as follows:

```
Menu
RootMenu
{
"Workspace      f.title
Menu"
"New Window"    f.exec "mterm"
"Mail"          f.exec "mail"
"Editor"        f.exec "editor"
"Refresh"       f.refresh
no−label        f.separator
"Restart"       f.restart
}
```

Once you have defined the menu, you need to bind the menu to a mouse button in your `.mwmrc` file:

```
<Btn3Down>  root f.menu RootMenu
```

Modifying Mouse Button Bindings

As described in Chapter 2, the Motif Window Manager recognizes the following button actions:

Press

Holding down a mouse button

Release

Releasing a pressed mouse button

Click

Pressing and releasing a mouse button

Double−click

Pressing and releasing a mouse button twice in rapid succession

Drag

Pressing a mouse button and moving the pointer/mouse device

You can associate a mouse button action with a window management function by using a button binding. A button binding is a command line you put in your `.mwmrc` file that associates a button action with a window manager function.

User−defined button bindings are added to built−in button bindings and are always defined first.

Default Button Bindings

The Motif Window Manager provides default button bindings. These button bindings define the functions of the window frame components. The user−specified button bindings that are defined with the *buttonBindings* resource are added to the built−in button bindings. The default value for this resource is *DefaultButtonBindings* (see Table 6−2).

Table 1 Default Button Bindings

Button Action	***Context***	***Function***
`Btn1Click2`	*menu*	**f.kill**
`Btn1Click`	*minimize*	**f.minimize**
`Btn1Click`	*maximize*	**f.maximize**
`Btn1Down`	*title*	**f.move**
`Btn1Down`	*window*|*icon*	**f.focus_key**
`Btn1Down`	*border*	**f.resize**
`Btn1Click`	*icon*	**f.post_wmenu**
`Btn1Click2`	*icon*	**f.restore**

Button Binding Syntax

The syntax for button bindings is as follows:

```
Buttons ButtonBindingSetName
{
button context   [| context ] function   [argument ]
button context   [| context ] function   [argument ]
...
button context   [| context ] function   [argument ]
}
```

Each line identifies a certain mouse button action, followed by the context in which the button action is valid, followed by the function to be done. Some functions require an argument.

Modifying Button Bindings

To modify the default button bindings, you need to edit either `system.mwmrc` to make system−wide changes or `.mwmrc` to make changes to the local environment. The easiest way to modify button bindings is to change the default bindings or to insert extra lines in the *DefaultButtonBindings*.

When modifying or creating a button binding, you need to first decide which mouse button to use and which action is performed on the button. Make sure you do not use button−action combinations already used by Motif. You might want to require a simultaneous key press with the mouse button action. This is called modifying the button action. Modifiers increase the number of possible button bindings you can make (see Table 6−3).

> **Note:** Binding a function to a mouse button−down event in a window, as would be done in the following example, has some undesirable side effects.
>
> ```
> <Btn1Down>   window          f.raise
> ```
>
> Once this binding is made, double clicking of that button on a PushButton inside a window will not be interpreted as a double−click, and the default PushButton action will not be taken. Therefore, when rebinding these events, it is important to keep the context of the rebinding as constrained as possible.

Table 2 Button Binding Modifier Keys

Modifier	***Description***
Ctrl	Control Key
Shift	Shift Key
Alt	Alt (Meta) Key
Lock	Lock Key
Mod1	Modifier 1
Mod2	Modifier 2
Mod3	Modifier 3
Mod4	Modifier 4
Mod5	Modifier 5

On some systems, you can bind up to five buttons if you have a 3−button mouse. For example, Button 4 is the simultaneous press of Buttons 1 and 2. Button 5 is the simultaneous press of Buttons 2 and 3. Each button can be bound with one of four actions (see Table 6−4).

Table 3 Button Actions for Button Bindings

Button	***Description***
Btn1Down	Button 1 press
Btn1Up	Button 1 release
Btn1Click	Button 1 press and release
Btn1Click2	Button 1 double−click
Btn2Down	Button 2 press
Btn2Up	Button 2 release
Btn2Click	Button 2 press and release
Btn2Click2	Button 2 double−click
Btn3Down	Button 3 press
Btn3Up	Button 3 release
Btn3Click	Button 3 press and release
Btn3Click2	Button 3 double−click

After choosing the optional modifier and the mouse button action, you must decide under which context(s) the binding works (see Table 6−5).

Table 4 Contexts for Mouse Button Bindings

This context...	***For mouse action at this pointer position...***
root	Workspace (root window)
window	Client window
frame	Window frame (title and border)
icon	Icon
title	Title bar
border	Frame minus title bar
app	Application window (inside the frame)

The context indicates where the pointer must be for the button binding to be effective. For example, a context of *window* indicates that the pointer

must be over a client window or window frame for the button binding to be effective. The *frame* context is for the window frame around a client window (including the border and title bar), the *border* context is for the border part of the window frame (not including the title bar), the *title* context is for the title bar of the window frame, and the *app* context is for the application window or client area (not including the window frame).

The following is an example of a button binding. Imagine you have created your own *Graphics Projects* menu and you want to display the menu with a button action. You choose `Alt` as a modifier and *Btn3Down* as the button action. You decide the pointer must be on the workspace. The function name for posting a special menu is **f.menu** and the argument is the menu name *Graphics Projects*. The following line in the *DefaultButtonBindings* in your `.mwmrc` file creates the button binding:

```
Alt<Btn3Down>        root    f.menu  "Graphics Projects"
```

Making a New Button Binding Set

If inserting a new button binding into the *DefaultButtonBindings* set is not enough, you may need to make a complete new set of button bindings. To create a new button binding set, use the *DefaultButtonBindings* in your `.mwmrc` file as a model. After you have created a new button binding set, use the *buttonBindings* resource to tell the window manager about it.

The *buttonBindings* resource specifies a button binding set. The default value of the resource is *DefaultButtonBindings*. Use the following syntax for specifying the resource in your `.Xdefaults` file:

```
Mwm*buttonBindings: NewButtonBindingSetName
```

This line directs the window manager to use *NewButtonBindingSetName* as the source of its button binding information. The button bindings are assumed to exist in the file named by the *configFile* resource; the default is `.mwmrc`.

For example, suppose that you want to specify a completely new button binding set instead of inserting a line in the existing *DefaultButtonBindings* set. The following entry in your `.mwmrc` file creates a new button binding set:

```
Buttons GraphicsButtonBindings
{
<Btn3Down>  root f.menu "Graphics Projects"
}
```

The following line in your `.Xdefaults` file references the new button binding set:

```
Mwm*buttonBindings: GraphicsButtonBindings
```

To display the graphics menu, press Button 3 on the mouse when the pointer is on the workspace.

Keyboard Bindings

In a manner similar to mouse button bindings, you can bind window manager functions to keys on the keyboard by using keyboard bindings.

Default Keyboard Bindings

Motif has default key bindings. These key bindings are replaced with user−specified key bindings specified with the *keyBindings* resource. Table 6−6 lists the default key binding specifications.

Table 1 Default Keyboard Bindings

Keys	*Context*	*Function*
Shift<Key>Escape	*window*|*icon*	**f.post_wmenu**
Alt<Key>space	*window*|*icon*	**f.post_wmenu**
Alt<Key>Tab	*root*|*icon*|*window*	**f.next_key**
Alt Shift<Key>Tab	*root*|*icon*|*window*	**f.prev_key**
Alt<Key>Escape	*root*|*icon*|*window*	**f.circle_down**
Alt Shift<Key>Escape	*root*|*icon*|*window*	**f.circle_up**
Alt Shift Ctrl<Key>exclam	*root*|*icon*|*window*	**f.set_behavior**
Alt<Key>F6	*window*	**f.next_key transient**
Alt Shift<Key>F6	*window*	**f.prev_key transient**
Shift<Key>F10	*icon*	**f.post_wmenu**

Keyboard Binding Syntax

The syntax for keyboard bindings is as follows:

```
Keys KeyBindingSetName
{
key context [| context ] function [argument ]
key context [| context ] function [argument ]
...
key context [| context ] function [argument ]
}
```

Each line identifies a unique key press sequence, followed by the context in which that sequence is valid, followed by the function to be done. Some functions require an argument. Context refers to the location of the keyboard input focus when a key is pressed.

Modifying Keyboard Bindings

To modify the default keyboard bindings, you need to edit either `system.mwmrc` to make system−wide changes or `.mwmrc` to make changes to the local environment. The easiest way to modify keyboard bindings is to change the default bindings or to insert extra lines in the *DefaultKeyBindings.*

When modifying a keyboard binding, you need to decide which key you want to bind and which action the key performs. Then choose the context in which the key binding is to work (see Table 6−7).

Table 2 Contexts for Key Bindings

Use this context...	*When the keyboard focus is here...*
root	Workspace (root window)
window	Client window (includes frame, title, border, and application window)
icon	Icon

Note that if **f.post_wmenu** or **f.menu** is bound to a key, **mwm** automatically uses the same key for removing the menu from the screen after it has been popped up.

Suppose you wanted to eliminate a particular keyboard binding. To disable it, you can delete or comment out the appropriate line in your `.mwmrc` file. You comment out a line by placing an *!* (exclamation point) comment character at the beginning of the line. The following shows an example of a commented−out line in a `.mwmrc` file:

```
Keys  DefaultKeyBindings
{
!Shift<Key>Escape    icon|window     f.post_wmenu
Alt<Key>Tab      window          f.next_key
}
```

Making a New Keyboard Binding Set

With keyboard bindings, as with button bindings, you have the option of creating a whole new binding set. To do so, use the *DefaultKeyBindings* of your `.mwmrc` file as a model. After you have created the new keyboard binding set, use the *keyBindings* resource to specify a key binding set in your `.Xdefaults` file:

`mwm*keyBindings:` NewKeyboardBindingSetName

The default value for this resource is *DefaultKeyBindings.*

Setting up mwm's Virtual Desktop Panning

This section describes an example of how you can modify your `.Xdefaults` file and `.mwmrc` file to activate virtual panning in **mwm**. The **mwm** functions **f.pan**, **f.goto**, and **f.track_pan** need to be bound either to keys or to mouse buttons.

Editing .mwmrc for Virtual Desktop Panning

Add the following to your `.mwmrc` file to use virtual panning.

Ensure that the name of your button bindings match the name given for the resource value **Mwm*buttonBindings** in your `.Xdefaults` file.

```
Buttons MyButtonBindings
{
    <Btn2Down>          root                    f.menu GotoMenu
    Meta<Btn1Down>      root                    f.track_pan
}
```

Ensure that the name of your key bindings match the name given for the resource value **Mwm*keyBindings** in your `.Xdefaults` file.

```
Keys MyKeyBindings
{
    Meta<Key>Up        window|root   f.pan 0,−100
    Meta<Key>Down      window|root   f.pan 0,100
    Meta<Key>Left      window|root   f.pan −100,0
    Meta<Key>Right     window|root   f.pan 100,0
}
```

To set up a menu that lets you quickly switch to different locations on the virtual desktop, code it as follows. You can then restart your **mwm** application.

```
Menu GotoMenu
{
    Up−Left     f.goto 1500,1100
    Up          f.goto 0,1100
    Up−Right    f.goto −1500,1100
    Left        f.goto 1500,0
    Home        f.goto 0,0
    Right       f.goto −1500,0
    Down−Left   f.goto 1500,−1100
    Down        f.goto 0,−1100
    Down−Right  f.goto −1500,−1100
}
```

Editing .Xdefaults for Virtual Desktop Panning

You can add the following to your `.Xdefaults` file to keep **mwm** from moving all your windows back to the visible part of the screen at startup:

```
Mwm*positionOnScreen:  False
```

Note that it is important to set the *positionOnScreen* resource to **False** because **mwm** automatically repositions all off−screen windows back to the display screen on restart.

To set up a menu to go to preset positions on the virtual canvas, add the following:

```
Mwm*GotoMenu*numColumns:    3
Mwm*GotoMenu*packing:        PACK_COLUMN
Mwm*GotoMenu*orientation:    HORIZONTAL
Mwm*GotoMenu*alignment:      ALIGNMENT_CENTER
```

To ensure consistency with the `.mwmrc` file, you can add the following lines:

```
Mwm*buttonBindings:     MyButtonBindings
Mwm*keyBindings:        MyKeyBindings
```

To prevent the client and icon windows from moving, you can use the following lines:

```
Mwm*wsm.iconPinned:     True
Mwm*wsm.clientPinned:   True
Mwm*iconPinned:         True
```

Where **wsm** is the workspace manager, which controls aspects of the desktop beyond the confines of the windows actually visible on the screen.

Controlling Client Access to mwm Windows

In support of the **mwm** Client−Command Interface (CCI), the MWM resource file allows the user to control access of other clients (such as other window managers) to the **mwm** menus. Clients such as the workspace manager demo which is included with Motif in `/demos/programs/workspace/wsm`, make use of the CCI to insert commands into **mwm**'s Root and Window menus. When one of these commands is selected, **mwm** sends a message back to the inserting client specifying the selected command. This allows the end−user to access workspace manager commands through *Mwm*'s menus without using screen real estate for the workspace manager application.

When an application inserts commands using the CCI protocol, the commands are inserted by **mwm** at the end of the appropriate menu. The location of these inserted commands can be modified by making changes to the Mwm resource file. More specifically, the CCI command placement is controlled using the **f.cci** command in the menu specifications of the Mwm resource file.

When a client, such as the workspace manager demo, inserts commands using the CCI, it specifies a command using an internal command−name along with a command−label. The command−label is the string that is displayed in the corresponding button of the appropriate Mwm menu. If you wish to control the placement of this command, you must refer to this command in the **mwm** resource file using the command−name. In addition, each command−name reference must be delimited with brackets as follows:

`<` command−name `>`

To obtain the corresponsing command−name, see the documentation from the inserting application. This documentation should describe the underlying command−names for each command.

The workspace manager demo adds two commands to the Mwm root menu. These commands are "Hide Workspace Manager" and "Switch Workspace". To be more precise, the "Hide Workspace Manager" entry is actually a toggle entry which changes to "Show Workspace Manager" when the Workspace Manager is hidden. The command−names are **_WSM_HIDE_WSM** and **_WSM_SWITCH_WORKSPACE**. The *Switch Workspace* command is actually a command−set and appears in the root menu as a cascade menu with a list of rooms as its sub−menu. The placement of the commands can be controlled using the **f.cci** function in the **mwm** resource file. The following sample root menu definition shows how the positions of these commands may be changed:

```
Menu MyRootMenu
{
"Root Menu"     f.title
DEFAULT_NAME    f.cci <_WSM_HIDE_WSM>
DEFAULT_NAME    f.cci <_WSM_SWITCH_WORKSPACE>
"New Window"    f.exec "xterm &"
"Shuffle Up"    f.circle_up
"Shuffle Down"  f.circle_down
"Refresh"       f.refresh
"Pack Icons"    f.pack_icons
 no−label       f.separator
"Restart..."    f.restart
"Quit..."       f.quit_mwm
}
```

The example above will cause the two commands *Hide Workspace Manager* and *Switch Workspace* to be placed at the top of the root menu, below the title. They will not be inserted at the end of the menu.

The format of the **f.cci** command is the following:

label `[`mnemonic `]` `[`accelerator `]` **`f.cci`** `[`modifier`]` command−reference

The label, mnemonic, and accelerator specifications are the same as for the other Mwm functions. The optional modifiers are described later in this reference page.

The *command−reference* specifies which command−name or command−names are being refered to. The preceding example refers to a single command and to a single command−set. In the **wsm** demo, the command "Switch Workspace" cascades to a sub−menu that contains a list of rooms. It is possible to refer to the entries within this sub−menu by concatinating command−names in the *command−reference*. This can be done be inserting a period between command−name specifications.

The submenu for the *Switch Workspace* menu entry contains, by default, the entries *Room1*, *Room2*, *Room3*, and *Room4*. Fortunately, **wsm** uses the same command−name and command−label for the entries. We can modify the *MyRootMenu* specification above to include *Room1* in the *MyRootMenu* specification by concatenating the command−set−name **_WSM_SWITCH_WORKSPACE** with the command−name *Room1* as

in the following example:

```
Menu MyRootMenu
{
"Root Menu"     f.title
DEFAULT_NAME    f.cci <_WSM_HIDE_WSM>
DEFAULT_NAME    f.cci <_WSM_SWITCH_WORKSPACE>
"Home"          f.cci <_WSM_SWITCH_WORKSPACE>.<Room1>
"New Window"    f.exec "xterm &"
"Shuffle Up"    f.circle_up
}
```

In the preceding example, a menu entry referring to *Room1* would appear below the *Switch Workspace* entry. The label *Home* would be used instead of the default label specified by **wsm**.

Often, it is necessary to refer to multiple command−names. This is possible by using a '*' as a wild−card symbol in place of the command −name. For example,

```
<_WSM_SWITCH_WORKSPACE>.<*>
```

refers to all command−names in the **_WSM_SWITCH_WORKSPACE** menu.