CHAPTER 26

WebChat

OVERVIEW

WebChat is a useful CGI program that allows a number of people on the World Wide Web to talk to one another simultaneously. It differs from a BBS (bulletin board system), in which the messages are typically read hours or days after they are posted. The ability to chat on the Web can be a quick way to hold a "virtual meeting." Figure 26.1 shows an example of what WebChat looks like.

Although both WebChat and WebBBS store messages for other people to read, there is a major difference in how the user sees and posts messages. The BBS emphasizes long-term storage of messages, including statistical data such as the date and time the message is posted. The BBS also encourages users to branch out into different topics in "threads" of replies.

On the other hand, WebChat emphasizes the quick posting of small messages much like a conversation among a group of people. Dialogue is designed to flow swiftly in small, easily digested chunks. Additionally, because the topic is being discussed by everyone at the same time, there is little room for different people discussing many different things in the same chat session. Thus, there is no reason to keep track of different threads of conversation.

Figure 26.1 *An example dialogue in WebChat.*

Because people are discussing things simultaneously in real time, another feature of WebChat is the ability to refresh or display new messages as quickly as desired. This is done using the META HTML tag to force refreshes within a certain time frame.

WebChat includes many features designed to facilitate this kind of dialogue. In WebChat, users can refresh messages using a button that is

displayed in plain view. In addition, if the user is using a browser such as Netscape, that supports the `META REFRESH HTML` tag, the user can choose to have the chat messages refresh or redisplay themselves automatically at a user-defined interval.

Messages are displayed in chronological order of posting from most recent to oldest so that users can quickly look through a list of statements. In addition, users can specify whether to see only new messages each time they refresh the screen or to include a user-defined number of previous messages. Viewing several of the previous posts along with new ones tends to provide the user with greater continuity.

By default, messages are posted to everyone, and the user's information is embedded as part of a posted message. This arrangement facilitates quick posting. By default, posted messages are seen by everyone. However, the user has a choice of entering a different username to specify whom the message should go to; the message is then entered as a private message from one person to another. This is option analogous to someone whispering a comment to someone else in the middle of a larger meeting.

Additionally, Netscape-style frames are supported; messages are refreshed in one frame while the user types messages in another frame. This feature allows a user to set a relatively high refresh rate for seeing new messages, while leaving the message submission form intact while the user is typing a message. Figure 26.2 shows an example of WebChat with frames.

WebChat also has configurable options such as the automatic announcement of a user's entry into the chat area, allowing people to keep track of who is currently in the system. Also, when a person leaves, he or she is encouraged to announce the departure by pressing the **Log Off** button. Nothing is more disturbing than to find out the person you were chatting with has left the room!

In addition, WebChat can be customized to remove old messages by age and by number of messages. For example, if WebChat is used for real-time conversations, it is generally not useful to keep the conversation messages for more than an hour. Additionally, you may want to make

sure that not more than 10 or 20 messages stay around at any given point, because messages older than the first 10 may be irrelevant to the current course of conversation. On the other hand, on other chat areas, you may want to keep the messages around for a long time to keep a full transcript of the discussion or meeting.

Figure 26.2 *WebChat with frames on.*

INSTALLATION AND USAGE

The chat files on the accompanying CD-ROM will install into a directory called **Chat**. The files and subdirectories associated with this application along with their required permissions are shown in Figure 26.3.

Chat is the root directory. It must be readable and executable by the Web server. In addition to the application files, the **Chat_open**, **Chat_CGI**,

and **Sessions** subdirectories are located here. Because **CGI-LIB.PL** is the only non–application-specific library that is used, it is stored in the main **Chat** directory along with the application.

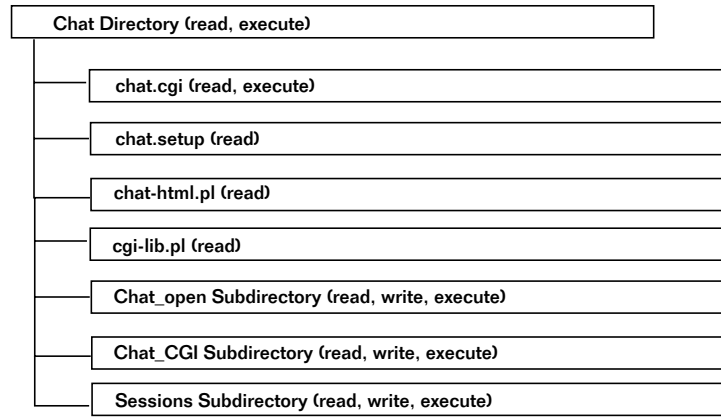

Figure 26.3 *Chat Script Directory Structure And Permissions.*

chat.cgi is the main CGI script that performs all the chat room functions, including displaying and posting new chat messages. This file must be readable and executable.

chat.setup is the setup file for the **chat.cgi** script. This file must be readable.

chat-html.pl contains Perl code that has various routines to output HTML related to the chat script. This file must be readable.

The **Sessions** subdirectory is used by the chat script to store the files related to each user's session after he or she first logs on to the chat room. This directory must be readable, writable, and executable.

The **Chat_open** subdirectory is used by **chat.cgi** to store messages for the Open Forum chat room. This is one of the chat rooms set up in the default **chat.setup** file. This directory must be readable, writable, and executable.

The **Chat_CGI** subdirectory is used by **chat.cgi** to store messages for the CGI talk chat room just as **Chat_open** does for the Open Forum . This directory must be readable, writable, and executable.

In addition to the chat messages, the various chat room directories also store "who" files that contain information about who is currently in each chat room. The chat script generates and deletes these files automatically, so you do not need to bother with maintaining them.

Server-Specific Setup and Options

The **chat.setup** file contains the configuration variables for **chat.cgi**. The following is a list of these setup items.

`@chat_rooms` is a list of chat room names. These names are descriptive names of the rooms that are available on the interactive chat. For example, if you had one chat room for discussing open topics and another for discussing CGI programming, this variable would be set to `("Open Chat Room", "CGI Programming Chat Room")`.

`@chat_directories` is an array that contains the directory names that match the list of rooms in `@chat_rooms`. Each of these directories stores only the chat messages related to its own corresponding room in the `@chat_rooms` array.

`@chat_variable` is a list of form variable names related to each chat room. Whenever **chat.cgi** is called after the initial logon, **chat.cgi** must have the variable `chat_room` sent to it. This variable should be equal to the name in the `@chat_variable` array. Each element of this array corresponds to each chat room listed in the `@chat_rooms` array. Because the values here are variable names, you should use lowercase text, underscores instead of spaces, and no special characters.

`$chat_script` is the **chat.cgi** script name. Most systems keep this variable set to `"chat.cgi,"` but some systems rename the script. For example, some Windows NT Web servers require the script name to be changed to a **.bat** extension. The `$chat_script` variable is used by **chat.cgi** to make references to itself from a URL.

`$no_html` is set to `on` to filter HTML out of a user's messages. It is a good idea to prevent users from posting HTML within their messages, because they can inadvertently do nasty things such as leave off a closure tag

(such as </H1> if they are including a header), extending the rogue tag to all the other messages. Figure 26.4 shows an example of how messages with HTML tags look after they are filtered.

Figure 26.4 *WebChat with HTML filtering activated.*

$no_html_images is set to on if you want to prevent people from referencing images in their messages. Setting $no_html to on also filters out image-related HTML tags, so setting this variable to on is for administrators who want to continue to allow HTML posting but not images. In other words, $no_html filters all HTML tags, including image tags, so $no_html_images need not be set if you have already configured $no_html to be on.

$chat_session_dir is the location of the directory where the session files are stored for chat users. When users log into the chat area, a session file is created so that they do not have to keep respecifying their information.

$chat_session_length is the time in days that session files stay active before being deleted. This value can be a fraction. For example, a value of .25 would delete sessions every quarter day (six hours).

Because Perl is actually processing the setup file, you can use a formula instead of a standard fractional value. A formula can be easier to read and maintain. For example, "1/24" (1 divided by 24) is a one-hour time frame. "1/24/12" (1 divided by 24 divided by 12) is a five-minute time frame.

$chat_who_length is the time in days that the who files stay active. Who files show who is active in a given chat room at a given time. This value can be fractional. Ideally, it should be very short. Using the value "1/24/12" (1 divided by 24 divided by 12) means that the who files stay around for about five minutes before being removed. A user can always "leave" a chat room by going to another WWW page on the Internet, and this act of leaving is not guaranteed to be sent to the chat script. If the who files are deleted often enough, they provide a relatively accurate way of determining who is currently in the system. Who files are refreshed whenever a user refreshes the chat messages or submits a message to the chat room.

$chat_announce_entry is on if you want a message to automatically post when someone enters a room. This message usually announces to everyone in the room that the user has logged on.

$prune_how_many_days is the number of days after which a chat message is considered too old to leave on the system. These messages are deleted. If this variable is set to zero, the chat messages will not be removed on the basis of age. This number may be fractional. For example, setting it to ".25" will delete messages older than six hours.

$prune_how_many_sequences is the maximum number of messages you want to leave on the system before the oldest ones are deleted. For example, if you specify this number to be 10, then only 10 messages will be allowed per chat room. In this case, after the 11th message is posted, message number 1 is deleted. Setting this value to zero means that you do not want any messages deleted on the basis of a maximum number of messages to keep on the system.

For a real-time chat system, it is recommended that you set up the system to keep few messages around. For one thing, in a real-time conversation, after about five or 10 minutes, people have probably moved on to another topic. Also, the chat script operates more efficiently if it does not have to process so many messages in the chat directory.

The following is an example of all the setup variables in the **chat.setup** file.

```
@chat_rooms = ("CGI Programming", "Open Forum");
@chat_room_directories = ("Chat_CGI", "Chat_Open");
@chat_room_variable = ("cgi", "open");

$chat_script = "chat.cgi";

$no_html = "off";
$no_html_images = "off";

$chat_session_dir = "Sessions";
$chat_session_length = 1;
$chat_who_length = 1/24/12;
$chat_announce_entry = "off";

$prune_how_many_days =          .25;
$prune_how_many_sequences =     10;
```

Using Other Setup Files

The chat script has the ability to reference another setup file in case the default **chat.setup** file does not meet the needs of every chat room. For example, although **chat.setup** accommodates multiple chat rooms, you may want to assign a different automatic removal of messages policy for each one. In the Open Chat Room, you may want to delete messages older than five minutes, but in the CGI programming chat room, you may not want to delete any messages.

You can do this by using another setup file that is loaded with the same variables defined in **chat.setup**. **chat.setup** is always loaded by the **chat.cgi** script. However, if you send the `setup` variable on the URL link to

the script as a Url-encoded variable, **chat.cgi** will read a setup file on the basis of that variable. For example, if you specified the call to **chat.cgi** as

```
http://www.foobar.com/cgi-bin/Chat/chat.cgi?setup=test
```

the **test.setup** file would be loaded by the chat script after the **chat.setup** file is loaded.

chat.setup is always necessary. This means that if you choose to override chat.setup using the `setup` form variable, you need specify only the variables you want changed in the new setup file instead of all the variables originally residing in chat.setup.

Modifying the HTML

The HTML used by the chat script is stored in the **chat-html.pl** file. This Perl script outputs the various HTML forms for viewing and posting chat messages. To modify the cosmetics of the chat script, you need only edit this file. The structure of this script is discussed in more detail in the "Design Discussion" section.

Because the chat script allows the user to choose frames versus a nonframes view of the chat script, the Perl code that generates the HTML for printing to the user's Web browser can seem a bit complex. If you plan to edit the HTML for the chat script, you should study the "Design Discussion" of chat-html.pl. Also, as usual, you should make a backup of any files you are planning to edit so that you can go back to the original if anything becomes messed up after something is changed.

Running the Script

To use **chat.cgi**, you simply call it by itself. A sample URL for this script, if it is installed in the **Chat** subdirectory underneath the **cgi-bin** directory, follows. The chat program automatically prints an HTML form asking

the user to log on to a chat room. An example of this form is displayed in Figure 26.5. However, if you are using a setup file other than **chat.setup**, you need to specify this in the URL that you use to run the chat script. An example of this alternative setup file was previously discussed.

```
http://www.foobar.com/cgi-bin/Chat/chat.cgi
```

Figure 26.5 *Example of chat room logon.*

CHAT ENTRANCE FORM VARIABLES

In the chat room logon screen shown in Figure 26.5, there are several form variables that the user is asked to fill out. These variables affect how the rest of the chat session is processed.

Username is the alias that the user will be referred to in the chat room. This field must be entered.

Email address is the E-mail address of the user. This is optional. If the user chooses to fill in this variable, a `MAILTO` hypertext reference tag will be displayed whenever the user's name is shown in the chat room.

Home page is the URL of the home page of the user. This field is also optional. If the user chooses to give the home page URL, a hypertext reference will be displayed whenever the user's name is shown in the chat room.

How many old messages to display determines how many old messages are displayed along with the new messages whenever the chat messages are loaded. Generally, an overlap of about 10 old messages is good for maintaining the continuity of a conversation. If you see only new messages and if they refer to a topic discussed previously, it is harder for most people to visualize how the conversation is flowing. Seeing a couple of the old messages serves as a reminder. This is especially true in chat rooms where many topics might be discussed at once.

Refresh rate is the number of seconds before the browser automatically reloads the script to display the new messages. This field is useful only for browsers that support the `META` refresh tag. Setting this field to zero disables automatic refreshing of messages.

If the check box for using frames is turned on, Netscape-style frames will be used to display messages in one frame while the submission form for a post is displayed in another frame. An example of frames was shown in Figure 26.2.

The **chat room** variable allows the user to select the chat room to enter.

Design Discussion

The chat application performs all its functions inside **chat.cgi**. These operations include the listing of messages in a chat room as well as the creation of those messages. Depending on the value of incoming form variables, **chat.cgi** determines which procedure to perform. A basic flowchart of the Web chat features is shown in Figure 26.6.

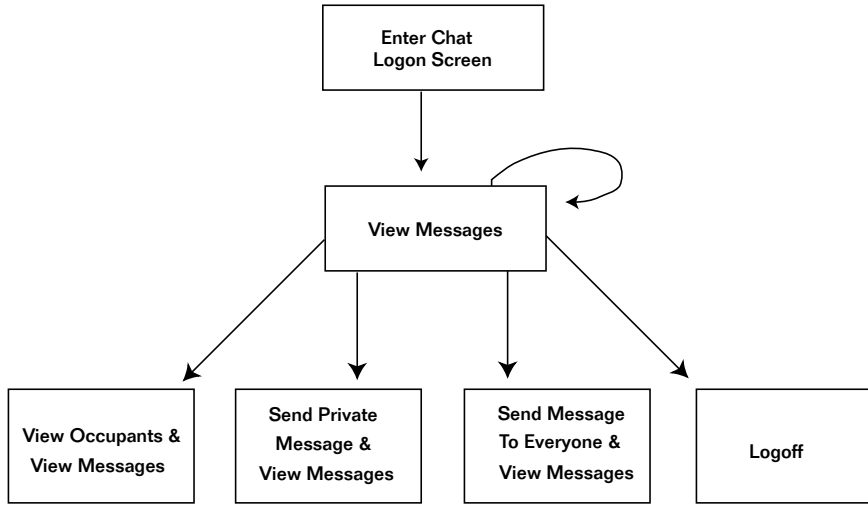

Figure 26.6 *Basic flow chart for the Web chat.*

Chat.cgi

The first line of the following code sets up the location of the supporting files to the program. By default, `$lib` is set to the current directory. Then the **cgi-lib.pl** library (for form parsing routines), the setup file for the script, and the Perl library containing the HTML code for the chat script are loaded.

```
$lib = ".";
require "$lib/cgi-lib.pl";
require "./chat.setup";
require "./chat-html.pl";
```

The incoming form variables are read to the `%in` associative array using the `ReadParse` subroutine.

```
&ReadParse;
```

As with the other CGI programs, the HTTP header is printed to let the Web server know that we are about to send HTML data to the Web browser. However, unlike most other CGI programs, Web Chat also prints a special code telling the Web browser not to cache the HTML during an active chat session. During a real-time chat, the user may reload the page every minute to look at the latest messages. It would be a waste of disk space to cache data that is constantly out of date. Thus, the `"Pragma: no-cache"` message is delivered along with the normal `"Content-type: text/html"` message.

The "no-cache" message is given only if a session form variable is set. This is because we want to cache the initial chat room entrance screen even if we do not cache the individual chat sessions. Because the messages are constantly changing, it is inefficient for the Web browser to constantly cache those pages. When the user first starts the script, no session variable has yet been set, and the script can use this fact to determine its course of action.

```
print "Content-type: text/html\n";

if ($in{'session'} ne "") {
    print "Pragma: no-cache\n\n";
} else {
    print "\n";
}
```

The form variables are read to regular Perl variables for easier processing later. `$chat_username` is the username of the user who is chatting. `$chat_email` is the E-mail address of the user. `$chat_http` is the URL that the user is associated with.

```
$chat_username = $in{'chat_username'};
$chat_email = $in{'chat_email'};
$chat_http = $in{'chat_http'};
```

`$refresh_rate` is the number of seconds before the browser automatically reloads the chat script to display the new messages.

```
$refresh_rate = $in{'refresh_rate'};
```

`$how_many_old` is a user-defined variable that determines how many old messages should be displayed along with the new messages.

```
$how_many_old = $in{'how_many_old'};
```

`$frames` is set to `on` if the user has chosen to use Netscape-style frames for interacting with other users in the chat room. With frames, the chat room is divided into two windows: one frame for viewing the messages and another frame for submitting new posts.

```
$frames = $in{'frames'};
```

If frames are currently being used, the script must figure out which frame it is currently being called from. If it is being called from the frame that displays messages, the `$fmsgs` variable is set to `on`. If the script is being called from the frame where messages are submitted, the `$fsubmit` variable is set to `on`. We need these variables in order to determine later whether the script should output the message list or the message submission form.

```
$fmsgs = $in{'fmsgs'};
$fsubmit = $in{'fsubmit'};
```

Figure 26.7 shows how **chat.cgi** is called when frames are activated. Figure 26.2 shows an example of the frames' output. When frames are activated, an HTML page that sets up the frames is printed by **chat.cgi**. This main frame HTML code sets up a top frame that contains messages and a bottom frame that contains the message submission form. As indicated previously, **chat.cgi** outputs the main frame HTML when the form variable `frame` is `on`. Then **chat.cgi** is called once for each of the two frames. When the form variable `fmsgs` is set to `on`, **chat.cgi** outputs the messages frame; when the form variable `fsubmit` is set to `on`, **chat.cgi** outputs the message submission frame.

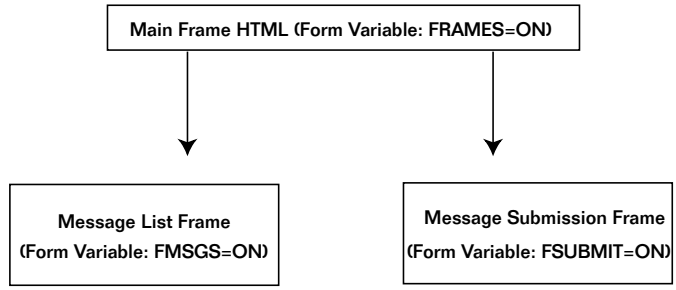

Figure 26.7 *Structure of frames in chat.cgi.*

`$user_last_read` stores the last read message relative to each user in the chat room. Because we want only new messages to be shown to the user (plus maybe a few old ones for continuity in the conversation), we keep track of the user's last read message number. The messages are created using ascending sequence numbers, so only numbers greater than the `$user_last_read` variable will be displayed. By default, `$user_last_read` is set to zero by the script. `$user_last_read` will be used later in the script when messages are being processed.

```
$user_last_read = 0;
```

`$chat_room` is set to the current chat room variable name. `$setup` is set to an alternative chat room setup filename. After this, if the alternative setup file is defined, it is also loaded by the chat script.

```
$chat_room = $in{'chat_room'};
$setup = $in{'setup'};

if ($setup ne "") {
    require "$setup.setup";
}
```

The chat script name is placed in the `$chat_script` variable. If this variable is not defined, it becomes `"chat.cgi"` by default. This variable should be defined in the **chat.setup** file if you are planning to change the name of the script. Generally, the only reason you would want to change

the name is if your Web server does not support the **.cgi** extension. Some Windows NT Web servers fall into this category.

```
if ($chat_script eq "") {
    $chat_script = "chat.cgi";
}
```

`$enter_chat` is set to the value of the **Enter Chat Room** button on the initial login HTML form. This value will be used later by **chat.cgi** to see whether the user has just entered the chat room and must be set up by the script.

```
$enter_chat = $in{'enter_chat'};
```

The following routine sets up variables from incoming form data as the result of a button being pressed on the Submit Chat Message form. `$refresh_chat`, `$submit_message`, `$logoff`, and `$occupants` are set to the value of their corresponding button labels if they were pressed. Only one of these variables will have a value associated with it, because only the pressed button has its value transferred as an incoming form value. This fact will be used later by **chat.cgi** to determine which operation to perform.

```
$refresh_chat = $in{'refresh_chat'};
$submit_message = $in{'submit_message'};
$logoff = $in{'logoff'};
$occupants = $in{'occupants'};
```

If a message is currently being submitted, the values of `$chat_to_user` and `$chat_message` are set by the incoming form variables. `$chat_to_user` defines the user to whom a chat message is directed. `$chat_message` is the chat message itself.

```
$chat_to_user = $in{'chat_to_user'};
$chat_message = $in{'chat_message'};
```

`$session` is set to the current session number. When users log in to a chat room, they are assigned a session number that **chat.cgi** uses to track their user information as well as their last read message number.

```
$session = $in{'session'};
```

By default, `$new_session` is set to `no`. This variable will be used later by the script to determine whether certain files still need to be set up for the newly logged in user.

```
$new_session = "no";
```

If the session has not yet been defined, then one of two things happens. If the user has seen the chat room logon screen, a session is created and the script continues processing. If the user has not yet seen the chat room logon screen, this HTML form is printed.

To see whether the user has been to the logon screen, the script checks the `$chat_username` variable. Remember that the `$chat_username` variable corresponds to the incoming `username` form variable. If this variable is not set, it is assumed that the user either has not entered all the information on the chat logon screen or has not been there yet. The script checks the `$enter_chat` variable. Again, recall that `$enter_chat` is set to a value if the **Enter Chat Room** button was pressed on the logon form. Thus, if `$enter_chat` has a value but `$chat_username` has none, the script prints the chat room logon screen using the `PrintChatEntrance` subroutine. It also prints an error message asking to the user to enter a username. Otherwise, the logon screen for the chat is simply displayed to the user.

```
if ($session eq "") {
    if ($chat_username eq "") {
        if ($enter_chat eq "") {
            &PrintChatEntrance($setup,"");
        } else {
            &PrintChatEntrance($setup,
             "Hey! You did not " .
             "enter a username.");
        }
    exit;
    }
```

A new session ID is created if no session ID is currently defined for the user and if the user already has a username. First, the `$new_session` vari-

able is toggled to yes. Then the new session ID is created and assigned to $session using the MakeSessionFile subroutine. This subroutine places all the logon information in a file for future reference by the chat script.

Notice in the following code that the last parameter is a zero. This value is the last read message number for the user. In other words, the user's session is initialized so that all the messages in the chat room are currently "new."

```
    $new_session = "yes";
    $session =
     &MakeSessionFile($chat_username, $chat_email,
        $chat_http, $refresh_rate,
        $how_many_old, "0");
}
```

Although we assigned the chat room name to the $chat_room variable, the script still must obtain the descriptive name of the chat room as well as the directory containing the chat messages. It uses the GetChatRoomInfo subroutine.

```
($chat_room_name, $chat_room_dir) =
 &GetChatRoomInfo($chat_room);
```

GetSessionInfo is called to retrieve information about the user currently being served by the chat script. Frame information ($fsubmit and $frames) is also submitted to GetSessionInfo because it normally updates the user's last read message count. However, if the chat script is currently outputting the submit message frame ($fsubmit) or outputting the main HTML frame document ($frames), then we do not update the user's last read message count.

A frame not related to message output may send a call to the script. If such a call updates the user's last message count, then another call—such as a call to the script where the messages are displayed in a different frame—will not display new messages. That's because the user's last read count has already been adjusted by the first frame. To avoid this problem, we send information to GetSessionInfo that specifies whether the script will output information to the user.

```
 ($user_name, $user_email, $user_http,
 $refresh_rate, $how_many_old,
 $user_last_read, $high_message) =
          &GetSessionInfo($session, $fsubmit, $frames);
```

If `$new_session` is yes and if `$chat_announce_entry` has been set to on in **chat.setup**, then variables are set up to generate an automatic chat message informing everyone of the user's chat room entrance. Figure 26.8 shows an example of an automatic logon message.

```
if ($chat_announce_entry eq "on" &&
    $new_session eq "yes") {
    $submit_message = "on";
    $chat_to_user = "ALL";
    $chat_message = "Automatic Message: $user_name
                     Joined Chat Room";
}
```

Figure 26.8 *Automatic logon message.*

If the logoff button was pressed, an automatic message is generated letting everyone know that the user has left the chat room. We use the same method that we used to generate the automatic chat entrance message. Figure 26.9 shows an example of the automatic logoff message.

```
if ($logoff ne "") {
    $submit_message = "on";
    $chat_to_user = "ALL";
    $chat_message = "Automatic Message: $user_name
                     Logged Off";
}
```

Figure 26.9 *Example of automatic message for logoff.*

You cannot really log off in a connectionless environment. However, the logoff button exists to help make it clear who is in the chat room. It works if everyone follows the etiquette of pressing the logoff button before moving to another Web page on the Internet.

The following routine reformats the date and time parts of the `localtime(time)` command into a `$current_date_time` variable. This variable will be used later to associate a nicely formatted date and time with each posted message.

```
 ($min, $hour, $day, $mon, $year) =
    (localtime(time))[1,2,3,4,5];
$mon++;
if (length($min) < 2) {
    $min = "0" . $min;
}
$ampm = "AM";
$ampm = "PM" if ($hour > 11);
$hour = $hour - 12 if ($hour > 12);
$current_date_time =
    "$mon/$day/$year $hour:$min $ampm";
```

Submit Chat Message

The next part of the main script processes the submission of a chat message. If the `$submit_message` button was pressed, the submission process begins.

```
if ($submit_message ne "") {
```

The next `if` statement checks to see whether `$chat_to_user` is addressed to no one (blank), `"all"`, or `"everyone"` without regard to case. If the statement is true, `$chat_to_user` is set to `"ALL"`. `"ALL"` is used to define the message as being posted to everyone in the chat room. If an explicit user name is given, then only the addressee to and the user who posted the original message can see the post.

```
    if ($chat_to_user eq "" ||
        $chat_to_user =~ /^all$/i ||
        $chat_to_user =~ /everyone/i) {
        $chat_to_user  = "ALL";
    }
```

First, we obtain the highest message number. Each time a post is made, the message number is incremented by 1. This arrangement keeps the message filenames unique and also gives us a way of to check whether a user has seen a message. If the user's last read number is less than a given message number, the script knows that the user has not yet read that message.

```
$high_number = &GetHighMessageNumber;
$high_number++;
```

The message number is formatted, using `sprintf`, into an integer with six characters. If the length of the number is less than six spaces, the leading spaces are converted to zeros using the `tr` function.

```
$high_number = sprintf("%6d",$high_number);
$high_number =~ tr/ /0/;
```

`sprintf` is used to format variables in various ways. In the chat script, `"%6d"` tells `sprintf` to format the `$high_number` variable as a decimal integer (`d`) with a length of six (`6`).

Next, the routine creates the new message file and writes all the fields to it. These fields include the username, E-mail address, URL link, the user the message is addressed to (usually `ALL`), the current date and time, and, of course, the chat message.

```
open(MSGFILE, ">$chat_room_dir/$high_number.msg");
print MSGFILE "$user_name\n";
print MSGFILE "$user_email\n";
print MSGFILE "$user_http\n";
print MSGFILE "$chat_to_user\n";
print MSGFILE "$current_date_time\n";
print MSGFILE "$chat_message\n";
close(MSGFILE);
```

Whenever messages are posted, the script also calls `PruneOldMessages` to delete any messages that are old.

```
&PruneOldMessages($chat_room_dir);
```

Because a new message has been posted, the user's last read field in the session file must be increased to accommodate the new message. We do this by calling `GetSessionInfo` all over again. However, we do not want to lose track of the last read message from the last time this script was called by the user. Thus, `$old_last_read` is used to mark the `$user_last_read` message. Then, after `GetSessionInfo` is called, `$user_last_read` is set back to the old value.

```
    $old_last_read = $user_last_read;
    ($user_name, $user_email, $user_http,
     $refresh_rate, $how_many_old,
     $user_last_read, $high_message) =
       &GetSessionInfo($session, $fsubmit, $frames);
    $user_last_read = $old_last_read;
}
```

Read the Current Occupants List

When the occupants list is displayed to the user, it is displayed as part of the general `$chat_buffer`, which contains all the messages to display. Before the script starts filling `$chat_buffer`, it is cleared. Figure 26.10 shows an example of the final occupants list on a user's Web browser.

```
$chat_buffer = "";
```

Figure 26.10 *Example of the WebChat occupants list.*

If $occupants has a value, then the **View Occupants** button was clicked by the user. This action starts the collection of the list of occupants into a form that can be displayed via HTML.

```
if ($occupants ne "") {
```

The chat room directory is opened, and all the files ending in **who** are read to the @files array using grep to filter the results of the readdir command.

```
    opendir(CHATDIR, "$chat_room_dir");
    @files = grep(/who$/,readdir(CHATDIR));
    closedir(CHATDIR);
```

The occupants list header is appended to $chat_buffer, which contains the HTML output for the messages to be displayed. Then, if there are who files in the @files array, each file is checked using a foreach loop.

```
    $chat_buffer .= "<H2>Occupants List</H2><P>";
    if (@files > 0) {
     foreach $whofile (@files) {
```

Each who file is opened, and a single line is read to the $wholine variable. Because the fields in $wholine are pipe-delimited, the split command is used to separate the fields into elements of the @whofields array.

```
        open (WHOFILE,"<$chat_room_dir/$whofile");
        $wholine = <WHOFILE>;
        @whofields = split(/\|/,$wholine);
        close(WHOFILE);
```

A sample who file looks like the following:

```
Gunther|gb@foobar.com|www.foobar.com|7/2/96 5:17 PM
```

Different HTML code is generated based on whether all the @whofields have values. For example, if an E-mail address exists for the user ($whofields[1]) then a hypertext reference is generated with a "MAILTO" tag. Otherwise, the plain username is printed as HTML.

```
if ($whofields[1] ne "") {
 $chat_buffer .= qq!<A HREF=MAILTO:! .
     qq!$whofields[1]>!;
}
$chat_buffer .= $whofields[0];
if ($whofields[1] ne "") {
 $chat_buffer .= "</A>";
}
```

`$whofields[3]` contains the last date and time that the person viewed messages. Remember, the who file is regenerated every time messages are viewed or submitted.

```
$chat_buffer .= " last viewed msgs at ";
$chat_buffer .= $whofields[3];
```

`$whofields[2]` contains the URL link for the user. If the user has given that information, then HTML code is generated to show a hypertext link to that URL.

```
if ($whofields[2] ne "") {
 $chat_buffer .=
     qq! (<A HREF="$whofields[2]">! .
           qq!Home Page</A>)!;
}
```

The occupants list portion of the `$chat_buffer` HTML code is ended with a paragraph break (`<P>`).

```
     $chat_buffer .= "<P>";
}
```

If there were no occupants to be found (no who files found), then `$chat_buffer` is merely set to `"No Occupants Found."`

```
} else {
 $chat_buffer .= "No Occupants Found";
} # End of no occupants
```

Finally, $chat_buffer is ended with a footer stating that the end of the occupants list has been reached.

```
    $chat_buffer .=
     "<P><H2>End of Occupants List</H2><P>";
} # End of occupants processing
```

Process Chat Messages

The next part of the chat script processes the chat messages for display to the user. Here, there is one thing to take into consideration: frames. If frames are activated, the chat program should display messages only if it has been called upon to read messages ($fmsgs is on). If the main frame HTML document is being output ($frames is on) or if the message submit frame is being output ($fsubmit), the script will not enter the part of the script that processes messages.

```
if ($fmsgs eq "on" ||
    ($frames ne  "on" &&
     $fsubmit ne "on")) {
```

Process Who Files

Before the messages are collected, a new who file is generated indicating that the user has read new messages. First, the old who file is deleted using unlink. Then the who file is re-created and the user information is printed to it: $user_name, $user_email, $user_http, and $current_date_time.

```
$whofile = "$chat_room_dir/$session.who";
unlink($whofile);
open(WHOFILE, ">$whofile");
print WHOFILE "$user_name|$user_email|$user_http";
print WHOFILE "|$current_date_time\n";
close (WHOFILE);
```

The code deletes the file instead of writing over it, to ensure that the who file is assigned a different file creation time. A subroutine discussed later removes old who files on the basis of creation time. Because this script is meant to run on multiple operating system platforms other than UNIX, the file is deleted and re-created to ensure consistency on as many platforms as possible.

The `RemoveOldWhoFiles` subroutine is called to delete who files for users that have not read messages within the `$chat_who_length` period of time. `$chat_who_length` is a global variable that is specified in **chat.setup**.

```
&RemoveOldWhoFiles;
```

Read Chat Messages

To read the chat messages, the script can be configured to restrict the number of messages that are seen. Generally, users do not want to see all the old messages over and over again, so the chat script keeps track of the last read message of the user. When it is created, each message is assigned a unique sequence number in ascending order. Whenever a user reads all the messages in the chat room directory, the current highest message number is set to the user's last read message number. Then, the next time the script is called to view messages, the "last read" message number can be compared against the message numbers in the directory. If the message number is lower than the last read number, we know that the message is old.

The `$msg_to_read` variable is set up to reflect the preceding algorithm except that 1 is added to it. Later, when the message numbers are compared, we will use the greater than or equal to (>=) 0 operator to compare the current message numbers to the last read message number stored in `$msg_to_read`.

```
$msg_to_read = $user_last_read + 1;
```

Next, `$how_many_old` is subtracted from `$msg_to_read`. As a result, some old messages are displayed with the new ones. Remember that in the chat room logon screen, the user chooses how many old messages to display with new ones.

```
$msg_to_read -= $how_many_old;
```

If there are fewer messages in the directory than `$how_many_old` messages, `$msg_to_read` could become a negative number or zero and the chat script would spend extra work trying to read files that do not exist. Thus, the next piece of code converts `$msg_to_read` into a positive value.

```
if ($msg_to_read < 1) {
    $msg_to_read = 1;
}
```

The next `if` statement checks quickly to see whether `$msg_to_read` is greater than the current `$high_message` number. If `$msg_to_read` is greater than `$high_message`, we do not need to bother iterating through all the files in the directory, because nothing would be displayed.

```
if ($high_message >= $msg_to_read) {
```

Now we begin reading the messages within the `for` loop started here.

```
    for ($x = $high_message; $x >= $msg_to_read; $x–){
```

The `sprintf` command is used to format the current message number, `$x`, to an integer with a length of `6`. Because `$x` is usually fewer than six characters long, `sprintf` pads it with leading spaces. Immediately afterward, the `tr` command is used to convert all the leading spaces to zeros. Thus, `sprintf` converts a number such as `"5"` to `"     5"`, and the `tr` command converts `"     5"` to `"000005."` This is done because the messages are stored in the chat room directory as six-digit numbers with leading zeros.

```
        $x = sprintf("%6d",$x);
        $x =~ tr/ /0/;
```

The message is checked for existence using the `-e` operator. If it exists, it is opened. If the opening of any message fails, the program exits with an error message printed to the user's Web browser.

```
        if (-e "$chat_room_dir/$x.msg") {
            open(MSG,"$chat_room_dir/$x.msg") ||
              &CgiDie("Could not open $x.msg");
```

If the file is opened successfully, the message is processed. Each line of the message corresponds to a field of information. The format of a message appears below:

```
[USERNAME OF USER WHO POSTED MESSAGE]
[EMAIL OF USER WHO POSTED MESSAGE]
[URL LINK TO USER WHO POSTED MESSAGE]
[USERNAME OF USER MESSAGE IS ADDRESS TO (USUALLY ALL)]
[MESSAGE DATE AND TIME]
[MESSAGE BODY]
```

All the preceding fields (except the message body) are read to the following variables: `$msg_from_user`, `$msg_email`, `$msg_http`, `$msg_to_user`, and `$msg_date_time`. The `<MSG>` command is a Perl convention that takes any file handle surrounded by brackets (<>) and returns the next line in the file. In addition, any fields that the user has entered are processed using the `HtmlFilter` function to remove HTML codes if you have disallowed them in **chat.setup**.

```
$msg_from_user = <MSG>;
$msg_from_user = &HtmlFilter($msg_from_user);
$msg_email = <MSG>;
$msg_email = &HtmlFilter($msg_email);
$msg_http = <MSG>;
$msg_http = &HtmlFilter($msg_http);
$msg_to_user = <MSG>;
$msg_to_user = &HtmlFilter($msg_to_user);
$msg_date_time = <MSG>;
```

The last character of all the variables is chopped, because a superfluous newline character is always appended to the end of a line read from a file.

```
chop($msg_from_user);
chop($msg_email);
chop($msg_http);
chop($msg_to_user);
chop($msg_date_time);
```

Messages are displayed to a user only if the addressee is `"ALL,"` if the addressee matches the current username, and if the poster username matches the current username.

```
if ($msg_to_user eq "ALL" ||
    $msg_to_user =~ /^$user_name$/i ||
    $msg_from_user =~ /^$user_name$/i) {
```

The information about the message is then converted to HTML code to be displayed to the user. This code is placed in the `$chat_buffer` variable, just as the occupants list HTML code was placed there previously.

Each message header is formatted as an HTML table. The first field set up in the table is the `"From:"` field.

```
$chat_buffer .= "<TABLE>\n";
$chat_buffer .= "<TR><TD>";
$chat_buffer .= "From:</TD><TD>";
```

If there is an E-mail address associated with the user who posted the message, a hypertext reference to the user's address is placed in `$chat_buffer`. Otherwise, `$msg_from_user` is placed in `$chat_buffer`.

```
if ($msg_email ne "") {
  $chat_buffer .= qq!<A HREF=MAILTO:! .
                  qq!$msg_email>!;
}
$chat_buffer .= $msg_from_user;
if ($msg_email ne "") {
  $chat_buffer .= "</A>";
}
```

If `$msg_http` is defined, the user's URL link will be added as a hypertext reference in `$chat_buffer`.

```
if ($msg_http ne "") {
  $chat_buffer .= qq! (<A HREF="$msg_http">! .
                  qq!Home Page</A>)!;
}
```

Periodically, the `$chat_buffer` fields are delimited with the standard `<TR>`, `</TR>`, `<TD>`, and `</TD>` HTML tags.

```
$chat_buffer .= "</TD>\n";
$chat_buffer .= "\n<TD>";
```

If the current message number ($x) is greater than $user_last_read, then a New Msg tag is appended to the information header.

```
if ($x > $user_last_read) {
    $chat_buffer .= " (New Msg) "
}
```

The date and time of the message are added to $chat_buffer.

```
$chat_buffer .= " at $msg_date_time</TD>";
```

If the message was generated for a particular user, a tag is generated for the information header that lets the user know that this is a private message to him or her or that it is a private message that the user posted to someone else.

```
$chat_buffer .= "</TR>\n";
if ($msg_to_user =~ /^$user_name$/i ||
    ($msg_from_user =~ /^$user_name$/i &&
     $msg_to_user ne "ALL")) {
$chat_buffer .= "<TR><TD>";
$chat_buffer .= "Private Msg To:" .
                "</TD><TD>$msg_to_user</TD>" .
                "</TR>\n";
}
```

The table is closed using the </TABLE> HTML tag, and then the body of the message is read from the file inside <BLOCKQUOTE> HTML tags. Each line is filtered using the HtmlFilter subroutine. In addition, each line is printed with a
 tag to show a line break.

```
$chat_buffer .= "</TABLE>\n";
$chat_buffer .= "<BLOCKQUOTE>\n";
while(<MSG>) {
    $_ = &HtmlFilter($_);
    $chat_buffer .= "$_<BR>";
    }
```

When the message body is finished, the file is closed and the program loops back to process another file unless the current message number ($x) has reached $high_message_number.

```
            close(MSG);
            $chat_buffer .= "\n";
        }
        $chat_buffer .= "</BLOCKQUOTE>\n";
        } # End of IF msg is to all or just us
    }
}
# End of IF we are not in the submit msg frame
#    or simply printing the main frameset
#    document
}
```

Process Logoff

If the logoff button was pressed, the who file is deleted immediately using unlink. The user will no longer show up on the occupants list.

```
if ($logoff ne "") {
    $whofile = "$chat_room_dir/$session.who";
    unlink($whofile);
}
```

Because the Web is connectionless, it is possible to stay online after you "log off." The ability to "log off" is presented as a means to allow a group of people who follow chat room etiquette to gain an accurate picture of who is in the chat room. It is not a security measure.

Print the Chat Screen

The final part of the main **chat.cgi** program is the printing of the chat page. The logic contained in the `PrintChatScreen` subroutine is complex, because it must account for both frames and nonframes printing of the chat messages and submit message form. An example of a new message in the chat message frame appears in Figure 26.11.

```
&PrintChatScreen($chat_buffer, $refresh_rate,
                 $session, $chat_room, $setup,
                 $frames, $fmsgs, $fsubmit);
```

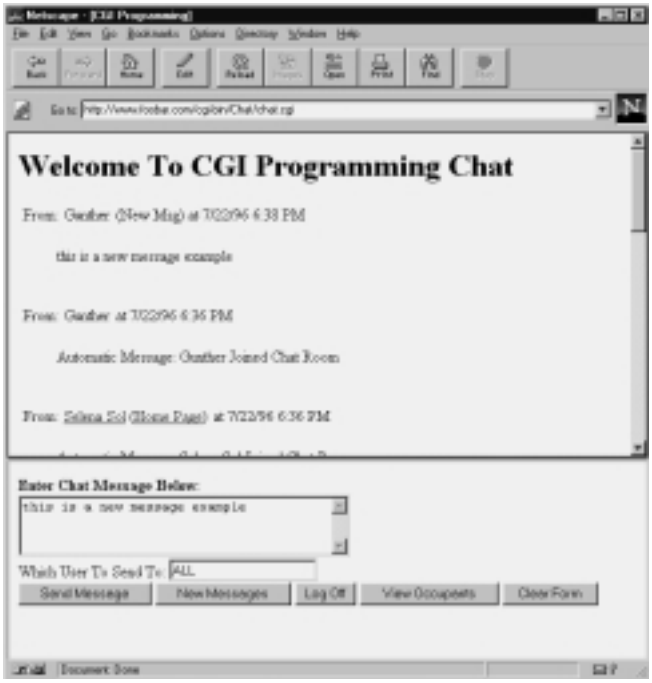

Figure 26.11 *Example of a new message appearing in the chat message frame.*

THE GETSESSIONINFO SUBROUTINE

The `GetSessionInfo` subroutine in the chat script retrieves information about the user's current session. This includes the username, E-mail address, URL link, refresh rate, use of frames, and last read message number. The routine starts by accepting the current session number and data about whether we are currently using frames (`$frames`) and, if so, whether we are refreshing the submit message frame (`$fsubmit`).

```
sub GetSessionInfo {
local($session, $fsubmit,$frames) = @_;
```

`$session_file`, `$temp`, `@fields`, `@f`, `$high_number`, and `$high_message` are declared local to the subroutine.

```
local($session_file);
local($temp,@fields, @f);
local($high_number, $high_message);
```

`$session_file` appends a **.dat** extension to the session ID. This is the name of the session file. Then the session file is opened in the **$chat_session_dir** directory.

```
$session_file = "$session.dat";

open (SESSIONFILE, "$chat_session_dir/$session_file");
```

The session file is read using a `while` loop. The session file should consist of one line of fields that are pipe-delimited. (The fields are separated by the pipe (|) symbol.) Here is an example session file:

```
Gunther|gb@foobar.com|http://foobar.com/|0|10|000067
```

The fields in the preceding session file are the username, E-mail address, URL, automatic refresh rate (in seconds), number of messages to display, and last read message number.

```
while (<SESSIONFILE>) {
$temp = $_;
}
```

A `chop` is added so that the last field does not have a hanging newline at the end. (A file read typically reads the whole line including the newline character that separates lines in a file.) The fields are separated into the `@fields` array. Finally, the session file is closed.

```
chop($temp);
@fields = split(/\|/, $temp);
close (SESSIONFILE);
```

`$high_message` is set to the highest message number in the current chat room. This message number is used to overwrite the last read message for the user. In other words, now that the user's last read message num-

ber has been read to `@fields` from the session file, we update it with the highest message number so that the next time the script runs, only the latest messages will be displayed to the user.

```
$high_message = &GetHighMessageNumber;
```

`@f` stores the contents of `@fields` as a temporary holder for the old session values. Then the last field in the `@fields` array is set equal to `$high_message`. This last field corresponds to the user's last read message number.

```
@f = @fields;
@fields[@fields - 1] = $high_message;
```

`@fields - 1` returns a number representing the number of the last element of the `@fields` array. `@fields` returns the total number of elements in the array, but because arrays start counting at zero instead of 1, we subtract 1 from `@fields` to get a reference to the last element of the `@fields` array. This technique is used throughout the chapter.

If frames are not on and if we are not printing the submit portion of a frame, the session file is updated with the new high message number. We open the session file and write the new `@fields` values to it.

```
if ($fsubmit ne "on" &&
    $frames ne "on") {
    open (SESSIONFILE,
      ">$chat_session_dir/$session_file");
    print SESSIONFILE join ("\|", @fields);
    print SESSIONFILE "\n";
    close (SESSIONFILE);
}
```

Finally, the original session values in `@f` are returned, along with the current high message number. This message number is returned because the script must know up to what sequence number to display new messages.

```
(@f, $high_message);

} # End of GetSessionInfo
```

Because the user's last read message number is set to `$high_message`, there is a chance that more messages might be posted to the chat room while this script is processing the messages to display. Although `$high_message` is actually out of date with regard to the true high message number, it is important to see that the user's last read message is in sync with the `$high_message` number of messages that have been displayed to the user.

GetHighMessageNumber Subroutine

The `GetHighMessageNumber` routine reads the current chat room directory and returns the highest message number found. It uses `$last_file` and `@files` as locally declared variables to do this processing.

```
sub GetHighMessageNumber {
local($last_file, @files);
```

First, `$chat_room_dir` is opened and all the files are read. The `grep` command is used to filter in only files that have **msg** in the filename. In addition, the messages are sorted.

```
opendir(CHATDIR, "$chat_room_dir");
@files = sort(grep(/msg/, readdir(CHATDIR)));
closedir(CHATDIR);
```

If the number of files in the array is greater than zero, the highest message number in the array is placed in `$last_file`. If there are no files with **msg** in the chat room directory, `$last_file` is set to zero. Note that the filenames are all six digits long and use leading zeros to pad the number.

```
if (@files > 0) {
    $last_file = $files[@files - 1];
} else {
    $last_file = "0000000";
}
```

The `substr` command is used to return the first six characters of the filename. Because the filenames contain six-digit numbers, this arrangement effectively returns just the numeric portion of the message filename.

```
substr($last_file,0,6);

} # End of GetHighMessageNumber
```

The MakeSessionFile Subroutine

The `MakeSessionFile` routine creates the session file that stores the current user information. `GetSessionInfo` later uses this session file to retrieve information about the currently logged on user every time the current session ID is sent to the routine. This routine starts by accepting a list of fields that make up the user information and then returns the newly acquired session ID that is associated with the session file.

```
sub MakeSessionFile {
local(@fields) = @_;
local($session, $session_file);
```

The first thing `MakeSessionFile` does is to call a routine (`RemoveOldSessions`) to remove old session files that are no longer being used. Then a new session ID is generated by generating a random number.

The random number is first seeded in the `srand` function by taking the value of the process ID and current time variable and combining them with the `or` operator (`|`). This random number, the time, and the process ID are converted to a long hexadecimal number that serves as the new session ID. A hexadecimal number is made up of digits that cover the numbers 0 through 9 and the letters A through F instead of the digits found in the base 10 system, 0 through 9. The session filename consists of the session ID plus a **.dat** extension.

```
&RemoveOldSessions;

srand($$|time);
$session = int(rand(60000));
$session = unpack("H*", pack("Nnn", time, $$, $session));

$session_file = "$session.dat";
```

Next, the session file is opened for creation in the directory specified by the `$chat_session_dir` variable, which has been set in **chat.setup**. The

user information fields are joined by the pipe symbol and are written to the file as a single line. Finally, the file is closed and the newly created session code is returned from the subroutine.

```
open (SESSIONFILE, ">$chat_session_dir/$session_file");
print SESSIONFILE join ("\|", @fields);
print SESSIONFILE "\n";

close (SESSIONFILE);

$session;
} # End of MakeSessionFile
```

The RemoveOldSessions Subroutine

The `RemoveOldSessions` procedure goes into the **$chat_session_dir** directory and removes all files that are older than `$chat_session_length`. These variables are set up in **chat.setup**. The `@files` array is used to contain all the filenames in the current directory. `$file` is a temporary variable used to hold the filename of the current file that the program is checking for age.

The directory is opened using the `opendir` command, and the files in the directory are read to an array using the `readdir` command. The output from `readdir` is passed to Perl's internal `grep` function to make sure that the special filenames "." and ".." escape the removal process.

```
sub RemoveOldSessions
{
local(@files, $file);

opendir(SESSIONDIR, "$chat_session_dir");
@files = grep(!/^\.\.?$/,readdir(SESSIONDIR));
closedir(SESSIONDIR);
```

The age of each file is then checked using the `-M` (modification date) operator. This operator returns the age of the file in days. If this age is greater than `$chat_session_length`, the `unlink` function is called to delete the file.

```
foreach $file (@files)
        {
```

```
# If it is older than session_length, delete it
        if (-M "$chat_session_dir/$file" >
            $chat_session_length)
                {
                unlink("$chat_session_dir/$file");
                }
        }
} # End of RemoveOldSessions
```

The RemoveOldWhoFiles Subroutine

`RemoveOldWhoFiles` takes who files in the current chat directory and checks to see whether they are old enough to expire. If they are, they are deleted. `@files` and `$file` are declared as local variables that are used throughout the routine processing.

```
sub RemoveOldWhoFiles
{
local(@files, $file);
```

The chat room directory is opened for reading by using the value stored in `$chat_room_dir`, a global variable that corresponds to the current chat room directory.

```
opendir(CHATDIR, "$chat_room_dir");
```

The filenames are read into the `@files` array, and the `grep` function is used to restrict these filenames to those that end in **who**.

```
@files = grep(/who$/,readdir(CHATDIR));
closedir(CHATDIR);
```

The body of the routine goes through each filename in the `@files` array.

```
foreach $file (@files)
        {
```

If the file in the **$chat_room_dir** directory is older than `$chat_who_length`, the file is deleted using the `unlink` command. When all the files have been checked, the subroutine exits.

```
        if (-M "$chat_room_dir/$file" >
            $chat_who_length)
                {
                unlink("$chat_room_dir/$file");
                }
        }
} # End of RemoveOldWhoFiles
```

The GetChatRoomInfo Subroutine

`GetChatRoomInfo` takes the chat room variable name (`$chat_room`) and returns the full descriptive name of the chat room as well as the directory where the chat room messages are stored.

```
sub GetChatRoomInfo {
   local($chat_room) = @_;
```

`$chat_room_name`, `$chat_room_dir`, `$x`, `$chat_room_number`, and `$error` are defined as local variables that will be used later in the subroutine.

```
   local($chat_room_name, $chat_room_dir, $x);
   local($chat_room_number, $error);
```

Initially, `$chat_room_number` is set to –1. At the end of the routine, the script will know that the name was not found in the list of chat room names if `$chat_room_number` is still –1. `$chat_room_number` will be set to the number of the element in the `@chat_room_variable` array in which the name of the chat room is defined if it exists.

```
$chat_room_number = -1;
```

The body of the `GetChatRoomInfo` routine uses a `for` loop to step through each element in the `@chat_room_variable` array.

```
for ($x = 1; $x <= @chat_room_variable; $x++)
        {
```

If the current element is equal to the contents of `$chat_room`, then `$chat_room_number` is set to the number of the current element in the array and the `for` loop exits when it encounters the `last` command.

```
        if ($chat_room_variable[$x - 1] eq $chat_room)
                {
                $chat_room_number = $x - 1;
                last;
                }
        } # End of FOR chat_room_variables
```

Now that the array has been processed, `$chat_room_number` should no longer be –1. If it is not –1, then `$chat_room_name` and `$chat_room_dir` are assigned their respective values based on the corresponding elements in the `@chat_rooms` and `@chat_room_directories` arrays.

```
if ($chat_room_number > -1) {
    $chat_room_name = $chat_rooms[$chat_room_number];
    $chat_room_dir = $chat_room_directories[$chat_room_number];
```

If `$chat_room_number` is still –1, then `$chat_room_name` and `$chat_room_dir` are cleared. To generate a better error message, `$chat_room` is set to `"None Given"` if `$chat_room` is an empty string. `$error` is set to a message telling the user that the `$chat_room` was not available Then `PrintChatError` sends the error message to the user, and the program exits with the `die` command.

```
} else {
    $chat_room_name="";
    $chat_room_dir = "";
    $chat_room = "None Given" if ($chat_room eq "");
    $error =
        "<strong>Chat Room: '$chat_room' Not
         Found</strong>";
    &PrintChatError($error);
    die;
}
```

If the routine successfully found that chat room information, it returns it as an array of two elements: `$chat_room_name` and `$chat_room_dir`.

```
($chat_room_name, $chat_room_dir);

} # end of GetChatRoomInfo
```

The PruneOldMessages Subroutine

The PruneOldMessages subroutine is responsible for removing old messages in a chat room directory.

```
sub PruneOldMessages {
```

$chat_room_dir is the only parameter that is sent to PruneOldMessages. It is declared local to PruneOldMessages. However, the global variables $prune_how_many_days and $prune_how_many_sequences affect how this routine deletes messages. These variables are defined in the setup file. $x, @files, and $prunefile are declared as local variables that will be used at various points during this subroutine.

```
    local($chat_room_dir) = @_;
    local($x, @files);
    local($prunefile);
```

The first major part of the routine reads all the filenames in the supplied chat room directory. The routine opens the directory and reads every filename that has a **msg** extension. These message filenames are sorted into the @files array.

```
opendir(CHATDIR, "$chat_room_dir");
    @files = sort(grep(/msg/, readdir(CHATDIR)));
    closedir(CHATDIR);
```

The routine then goes through each of the files in the @files array.

```
    for ($x = @files; $x >= 1; $x—) {
```

$prunefile is set to the full path and filename of the file that is currently being checked for age. The -M parameter is used to check the last modification date in days. If it is greater than $prune_how_many_days and if $prune_how_many_days is greater than zero, the file is deleted and the name is removed from the @files array.

```
        $prunefile = "$chat_room_dir/$files[$x - 1]";
        # First we check the age in days
        if ((-M "$prunefile" > $prune_how_many_days) &&
            ($prune_how_many_days > 0)) {
            unlink("$prunefile");
            &RemoveElement(*files, $x - 1);
            next;
        }
```

$x is the current number of the element that we are processing in the @files array. If $x is less than or equal to the total number of elements in the array minus the maximum number of sequences to keep around ($prune_how_many_sequences) and $prune_how-many_sequences is not zero, then the file is deleted and the corresponding element is removed from the @files array.

```
        if (
           ($x <= (@files - $prune_how_many_sequences))
            && ($prune_how_many_sequences != 0)) {
            unlink("$prunefile");
            &RemoveElement(*files, $x - 1);
            next;
        }
    } # End of for all files
} # End of PruneOldMessages
```

The RemoveElement Subroutine

The RemoveElement subroutine is simple. It takes a reference to an array and the number of the element to delete from the array and uses Perl's splice function to remove the element. Finally, the routine returns the resulting array.

```
sub RemoveElement
{
local(*file_list, $number) = @_;

if ($number > @file_list)
        {
        die "Number was higher than " .
            "number of elements in file list";
        }
```

```
splice(@file_list,$number,1);
@file_list;
} # End of RemoveElement
```

The HtmlFilter Subroutine

`HtmlFilter` is a function that takes a string and strips out all the HTML code in it depending on how the global variables `$no_html_images` and `$no_html` have been set.

```
sub HtmlFilter
{
```

`$filter` is a local variable that is assigned the string of characters that may contain HTML code to be filtered out.

```
local($filter) = @_;
```

`If $no_html_images` is `on`, then all HTML tags that contain `"IMG SRC"` have the brackets (<>) transformed into `"&LT"` and `"&GT"` tags, respectively. The HTML tags `"&LT"` and `"&GT"` are used to print "less than" and "greater than" symbols in the place of the brackets for the HTML tags.

```
if ($no_html_images eq "on")
{
    $filter =~ s/<(IMG\s*SRC.*)>/&LT;$1&GT;/ig;
} # End of parsing out no images
```

If `$no_html` is `on`, all HTML tags have their brackets (<>) transformed into `"&LT"` and `"&GT."`

```
if ($no_html eq "on")
{
    $filter =~ s/<([^>]+)>/\&LT;$1&GT;/ig;
} # End of No html
```

Finally, the subroutine returns the filtered text.

```
$filter;

} # End of HTML Filter
```

Chat-html.pl

Chat-html.pl contains the procedures that print the various HTML screens for **chat.cgi**. If you wish to modify the user interface or look-and-feel of the program, you will most likely find the target routine in this file.

The PrintChatEntrance Subroutine

`PrintChatEntrance` prints the original HTML form that logs the user into a chat room. It takes two parameters: `$setup` and `$chat_error`. If an error occurs in processing the user's logon information, the nature of the error is placed in `$chat_error`, and `PrintChatEntrance` is called again to make the user enter the correct information. `$setup` is passed so that the HTML form can pass a hidden input field with the alternative setup filename.

```
sub PrintChatEntrance {
local($setup,$chat_error) = @_;
```

`$chat_room_options` is declared as a local variable. It contains the list of descriptive names for all the chat rooms the user can enter.

```
local ($chat_room_options);
```

`$setup` is set to nothing if it is already set to the default setup file prefix, `"chat."`

```
$setup = "" if ($setup eq "chat");
```

`$chat_room_options` is built up as a string of all the HTML `<OPTION>` tags that go along with each chat room name.

```
$chat_room_options = "";
```

```
for (0..@chat_rooms - 1) {
$chat_room_options .=
  "<OPTION VALUE=$chat_room_variable[$_]>" .
  "$chat_rooms[$_]\n";
}

if ($chat_room_options eq "") {
    $chat_room_options =
        "<OPTION>Chat Room Not Set Up\n";
}
```

Finally, the main HTML form is printed using the HERE DOCUMEN" method. The $setup and $chat_room_options variables are included in the output. The output of this HTML code is shown back in Figure 26.5.

```
print <<__END_OF_ENTRANCE__;
<HTML>
<HEAD>
<TITLE>Chat Page</TITLE>
</HEAD>
<BODY>
<H1>Welcome To The Chat Page</H1>
<H2>$chat_error</H2>
<FORM METHOD=POST ACTION=chat.cgi>
<INPUT TYPE=HIDDEN NAME=setup VALUE=$setup>
<HR>
<STRONG>Enter Information Below:</STRONG><p>

<TABLE BORDER=1>
<TR>
<TD ALIGHT=RIGHT>User Name:</TD>
<TD><INPUT NAME=chat_username></TD>
</TR>
<TR>
<TD ALIGHT=RIGHT>Your Email Address(*):</TD>
<TD><INPUT NAME=chat_email></TD>
</TR>
<TR>
<TD ALIGHT=RIGHT>Your Home Page (*):</TD>
<TD><INPUT NAME=chat_http></TD>
</TR>
<TR>
<TD ALIGHT=RIGHT>How Many Old Messages To Display:</TD>
<TD><INPUT NAME=how_many_old VALUE="10"></TD>
</TR>
```

```
<TR>
<TD ALIGHT=RIGHT>Automatic Refresh Rate (Seconds):</TD>
<TD><INPUT NAME=refresh_rate VALUE="0"></TD>
</TR>
<TR>
<TD ALIGHT=RIGHT>Use Frames?:</TD>
<TD><INPUT TYPE=checkbox NAME=frames></TD>
</TR>
<TR>
<TD ALIGHT=RIGHT>Chat Room</TD>
<TD><SELECT NAME=chat_room>
$chat_room_options
</SELECT>
</TD>
</TR>
</TABLE>
<P>
<INPUT TYPE=SUBMIT NAME=enter_chat
VALUE="Enter The Chat Room">

<P>
<STRONG>Special Notes:</STRONG><P>
(*) Indicates Optional Information<P>
Choose <STRONG>how many old messages</STRONG> to display if you want
to display some older messages along with the new ones whenever you
refresh the chat message list.
<P>
Additionally, if you use Netscape 2.0 or another browser that supports
the HTML <STRONG>Refresh</STRONG> tag, then you can state the number
of seconds you want to pass before the chat message list is automati-
cally refreshed for you. This lets you display new messages automati-
cally.
<P>
If you are using Netscape 2.0 or another browser that supports
<STRONG>Frames</STRONG>, it is highly suggested that you turn frames
ON. This allows the messages to be displayed in one frame, while you
submit your own chat messages in another one on the same screen.

 <HR>
</FORM>
</BODY>
</HTML>
__END_OF_ENTRANCE__
} # end of PrintChatEntrance
```

The PrintChatScreen Subroutine

The PrintChatScreen routine is the heart of the chat program's HTML output. All the chat messages and message submission forms are printed in this subroutine. In addition, the routine also detects whether the user has chosen to use frames rather than one Web browser screen to display the messages and submission form.

PrintChatScreen accepts a variety of parameters. $chat_buffer contains the HTML code for the messages the user will see along with an occupants list if the user requested it. $refresh_rate is set if the user has chosen to use the META refresh tag to make the HTML page reload after a predetermined number of seconds. $session is the current session ID that **chat.cgi** uses to keep track of the user from screen to screen. $chat_room is the current chat room name. $setup is the alternative setup file name for **chat.cgi**.

$frames, $fmsgs, and $fsubmit are all related to processing frames. If $frames is on, PrintChatScreen is printing with frames. If $fmsgs is on, the script is currently printing the messages frame. If $fsubmit is on, the script is printing the frame with the message submission form. If neither $fsubmit nor $fmsgs is on and if $frames is on, the main frame HTML document that points to a message and a submission form frame is printed. $frames should be on only if the main frame HTML document is being sent to the user's Web browser.

```
sub PrintChatScreen {
local($chat_buffer,
      $refresh_rate, $session,
      $chat_room, $setup,
      $frames, $fmsgs, $fsubmit) = @_;
```

Several other variables are declared local to the subroutine. $chat_message_header will contain HTML code that will serve as a header for the chat messages if they are currently being printed. $chat_refresh will contain the HTML META refresh tag if $refresh_rate has been set to a value greater than zero. $more_url and $more_hidden will be used to keep tabs

on form variables, such as the name of the alternative setup file and the session ID, that must be passed from chat screen to chat screen.

```
local($chat_message_header, $more_url,
      $more_hidden, $chat_refresh);
```

If `$setup` is the prefix `"chat"` for the default setup file, **chat.setup**, the value of `$setup` is cleared. There is no need to pass unnecessary information about the default setup file from screen to screen.

```
$setup = "" if ($setup eq "chat");
```

As mentioned previously, `$more_url` and `$more_hidden` contain extra fields of information that is passed from chat screen to chat screen. `$more_hidden` formats these fields as hidden input fields on the HTML forms. `$more_url` is used to extend the URL that is used to call the **chat.cgi** script using the `META` refresh tag so that the URL includes the variables listed in `$more_hidden`.

```
$more_url = "";
$more_hidden = "";
if ($setup ne "") {
    $more_url = "&setup=$setup";
    $more_hidden = "<INPUT TYPE=HIDDEN NAME=setup " .
                   "VALUE=$setup>";
}
$more_url = "session=$session" .
               "&chat_room=$chat_room" .
          $more_url;
```

If `$refresh_rate` is a positive number, a `META` tag is generated to make the Web browser automatically reload the page after `$refresh_rate` seconds. The URL that is called has `$more_url` added to it so that certain variables, such as the session ID, are passed from script to script and hence from screen to screen.

```
if ($refresh_rate > 0) {
    $chat_refresh =
      qq!<META HTTP-EQUIV="Refresh" ! .
      qq!CONTENT="$refresh_rate; ! .
```

```
      qq!URL=chat.cgi?$more_url!;
```

In addition to `$more_url`, if `$frames` is currently `on` and if the messages frame is printing, then the `META` refresh tag must have `"$fmsgs=on"` added to the list of variables being sent.

```
    if ($frames ne "on" && $fmsgs eq "on") {
        $chat_refresh .= "&fmsgs=on";
    }
    $chat_refresh .= qq!">!;
} else {
    $chat_refresh = "";
}
```

The Perl `qq` command is used in several places here to change the default string delimiter from double quotes (") to an exclamation point (!). This technique is explained in more detail in Appendix A.

If `$fsubmit` is `on` and if the main `$frames` HTML document is not being printed, then `$chat_refresh` is cleared.

```
if ($frames ne "on" && $fsubmit eq "on") {
    $chat_refresh = "";
}
```

If `$frames` is `on`, the main HTML frame document is printed to the user's Web browser using the `HERE DOCUMENT` method. This document sets up the two frames and points to the **chat.cgi** script for printing the messages in one frame (`fmsgs=on`) and the submission form in another one (`fsubmit=on`).

```
if ($frames eq "on") {
    print <<__END_OF_MAIN_FRAME__;
<HTML>
<HEAD>
<TITLE>$chat_room_name</TITLE>
</HEAD>
<FRAMESET ROWS="*,210">
   <FRAME NAME="_fmsgs" SRC=chat.cgi?fmsgs=on&$more_url>
```

```
   <FRAME NAME="_fsubmit" SRC=chat.cgi?fsubmit=on&$more_url>
</FRAMESET>
</HTML>
__END_OF_MAIN_FRAME__
}
```

If the main frame document is not being printed, then the standard HTML header is output using the "here document" method.

```
if ($frames ne "on") {
print <<__END_OF_HTML_HEADER__;
<HTML>
$chat_refresh
<HEAD>
<TITLE>$chat_room_name</TITLE>
</HEAD>
<BODY>
__END_OF_HTML_HEADER__
}
```

If `$fsubmit` is `on`, the message submission frame is being printed. This means that the `<FORM>` tag should target the `"_fmsgs"` (message list) frame whenever information is submitted from the message submission form to the chat script. The target is set to the messages frame instead of the message submission frame; when a new message is submitted or another button, such as **View Occupants**, is pressed, we want the messages frame—and not the message submission frame—to be updated with the new messages.

```
if ($fsubmit eq "on") {
$form_header = <<__END_FORM_HDR__;
<FORM METHOD=POST ACTION=chat.cgi TARGET="_fmsgs">
__END_FORM_HDR__
```

If the submission frame is not being printed, a normal form header is derived that has no specific frame target.

```
} else {
$form_header = <<__END_FORM_HDR__;
<FORM METHOD=POST ACTION=chat.cgi>
__END_FORM_HDR__
}
```

Additionally, if the submission frame is being printed, the form header must include a hidden tag telling the script that it must refresh the messages frame (`fmsgs=on`).

```
if ($fsubmit eq "on") {
    $form_header .= qq!<INPUT TYPE=HIDDEN NAME=fmsgs! .
                    qq! VALUE=on>!;
}
```

If the messages frame is being printed, no form header should be generated.

```
if ($fmsgs eq "on") {
    $form_header = "";
}
```

By default, there is no chat message header. But if we are printing the message frame, we want a small header to print, so the `$chat_message_header` variable has a header assigned to it.

```
$chat_message_header = "";
if ($fmsgs ne "on") {
    $chat_message_header = "<H2>Chat Messages:</H2>";
}
```

If the message frame is being printed or if frames are not activated, a general chat screen header is printed using the `HERE DOCUMENT` method.

```
if (($frames ne "on" &&
     $fsubmit ne "on") ||
    $fmsgs eq "on") {
    print <<__END_OF_CHAT_HEADER__;
<H1>Welcome To $chat_room_name Chat</H1>
__END_OF_CHAT_HEADER__
}
```

If the message submission frame is being printed or if frames are not activated, then the submission form is printed to the user's Web browser.

```
if ($fsubmit eq "on" ||
        ($frames ne "on" && $fmsgs ne "on")) {
     print <<__END_OF_CHAT_SUBMIT__;
```

```
$form_header

<INPUT TYPE=HIDDEN NAME=session VALUE=$session>
<INPUT TYPE=HIDDEN NAME=chat_room VALUE=$chat_room>
$more_hidden
<STRONG>Enter Chat Message Below:</STRONG>
<BR>
<TEXTAREA NAME=chat_message
ROWS=3 COLS=40 WRAP=physical></TEXTAREA>
<BR>
Which User To Send To:
<INPUT TYPE=TEXT NAME=chat_to_user
VALUE="ALL">
<BR>
<INPUT TYPE=SUBMIT NAME=submit_message
VALUE="Send Message">
<INPUT TYPE=SUBMIT NAME=refresh_chat
VALUE="New Messages">
<INPUT TYPE=SUBMIT NAME=logoff
VALUE="Log Off">
<INPUT TYPE=SUBMIT NAME=occupants
VALUE="View Occupants">
<INPUT TYPE=RESET
VALUE="Clear Form">
</FORM>
__END_OF_CHAT_SUBMIT__
```

An extra HTML `<HR>` tag is printed to separate the submission form from the message list if frames are not used and if the submission form has just been output to the user's Web browser.

```
if ($fsubmit ne "on") {
    print "<HR>\n";
}
}
```

If the messages frame is being output or the frames feature is not being used, then the chat messages are printed (`$chat_buffer`) along with the chat message list header (`$chat_message_header`).

```
if (($frames ne "on" &&
     $fsubmit ne "on") ||
    $fmsgs eq "on") {
```

```
    print <<__END_OF_CHAT_MSGS__;
$chat_message_header
$chat_buffer
__END_OF_CHAT_MSGS__
```

Just as with the submission form, an extra `<HR>` is printed at the end of the message list if the frames feature is not being used.

```
if ($fmsgs ne "on") {
    print "<HR>\n";
}
}
```

Finally, the chat footer is printed and the subroutine ends.

```
    if ($frames ne "on") {
     print <<__END_OF_CHAT_FOOTER__;
</BODY>
</HTML>
__END_OF_CHAT_FOOTER__
}
} # end of PrintChatScreen
```

The PrintChatError Subroutine

`PrintChatError` prints any errors that have occurred in the **chat.cgi** program. It accepts only an `$error` parameter. The routine uses the contents of `$error` to store the nature of the error message. Figure 26.12 shows an example of an error occurring in the chat script.

```
sub PrintChatError {
local($error) = @_;

print <<__END_OF_ERROR__;
<HTML><HEAD>
<TITLE>Problem In Chat Occurred</TITLE>
</HEAD>
<BODY>
<h1>Problem In Chat Occurred</h1>
<HR>
<blockquote>
$error
```

```
</blockquote>
<HR>
</BODY></HTML>
__END_OF_ERROR__
} # End of PrintChatError
```

Figure 26.12 *Sample output for the PrintChatError subroutine.*