Hierarchical Distributed Repositories in Concurrent Versions System

by

Miroslav Jurišić

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

Author .....
Department of Electrical Engineering and Computer Science
Jan 19, 2001

Certified by .....–James D. Bruce
Vice President, Information Systems
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Hierarchical Distributed Repositories in Concurrent Versions System

by

Miroslav Jurišić

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 19, 2001, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a hierarchical distributed repository system for Concurrent Versions System (CVS). The hierarchical repository system is based on the existing CVS implementation. I modified the repository structure, client-server protocol, working files structure, and user interface to provide the facilities necessary to maintain information about a distributed repositoy.

Thesis Supervisor: James D. Bruce –Title: Vice President, Information Systems

Acknowledgments

This thesis is my last major work as an MIT student (at least for the foreseeable future), and as such it would not have been possible without those who helped me become an MIT student, and those who helped me as an MIT student. In addition to that, the thesis would have taken an entirely different shape without the influence of the people who helped me bring it to completion. To all of those people, I owe many thanks, even if I can't list them all. However, some of them deserve a special mention.

My mother's support for my interests was instrumental in bringing me to MIT. Without all the toys and books she gave me, I would not have discovered my most rewarding interests until too late.

I owe the most crucial seeds of my knowledge and enthusiasm to the people who nourished my interest in math and computer science, and poured their energy into my education: Ivo Melvan, Mea Bombarelli, Snježana Serdar, Ilko Brnetić.

Daria Rovan imbued me with a better understanding of who I am and who I want to become, and for that I will be forever grateful.

Dean Bonnie Walters provided more help in overcoming my culture shock than I ever imagined would be necessary. Her words and support were the root of my strength and entusiasm, and enabled me to stand up to the challenges I faced at MIT.

Marshall Vale, Scott McGuire, and Alexandra Ellwood provided me with an environment in which I could express my tendency to try to do too much all at once, and I thank them for all the effort they put into providing me with hard problems.

Jim Bruce stepped in at the moment of greatest uncertainty in my life and gave me a wonderful opportunity to do exactly what I wanted.

Greg Hudson reviewed the proposals, draft, and the thesis itself, improving them in countless ways.

Without Jelena's precious companionship and welcome distractions, many things I had dreamed of would have remained just that.

Kad već postoji planina . . .

Kad već postoji planina, treba se penjati
Strpljivo i dugo do samoga vrha.
Kad već postoji planina, treba je upoznati
Srcem i umom kao materinsku riječ.
Kad već postoji planina, treba je osvojiti
Polako i mudro kao jedinu ljubav.
Kad već postoji planina, treba je zavoljeti
Predano i nježno kao dijete.
Zato i postoji planina:
Da bismo otkrili
Pute neprohodne.

Ljerka Car Matutinović

Contents

Chapter 1

Introduction

CVS (Concurrent Version System) is a version control system, initially released in 1990 and based on RCS (Revision Control System) from 1982. Building on the basic version control primitives provided by RCS, CVS added many higher-level functions which simplified management of large projects, focusing on manipulating groups of files rather than individual files. CVS provides developers with the ability to track changes to a project over time, to revert parts of the project to their state at an arbitrary point in time, to work on the project concurrently with other developers, and to work on different variants of the project simultaneously. CVS operations provide relatively simple ways of performing most common manipulations on a large project, while still having the flexibility that allows fairly complex version control tasks.

CVS initially did not support client-server operation, but the support was added early on, making CVS very popular in academic environments, as well as many open-source projects. Its popularity increased even more with support for authentication and encryption, and ports to many UNIX platforms, Mac OS and Windows.

1.1 Motivations for hierarchical repositories

Unfortunately, several major features of CVS came about as an afterthought or a side-effect of the original implementation rather than through careful design. As a

result, CVS sometimes falls short of what developers expect from a version control system. One particular area in which CVS is limited is that there is no interaction between different source code repositories. Although there are relatively primitive facilities for manually propagating information between different repositories, their use is generally tedious and error-prone.

The goal of this thesis is to extend CVS and provide the ability to easily propagate information between repositories, possibly stored on different servers, and to establish interconnections between different repositories. These extensions represent the ideas that different groups of people might work on one project, without necessarily sharing the CVS server infrastructure, and that a single project might actively incorporate parts of other projects, without having to share the CVS infrastructure with them. The following situations, for example, might be represented by this model:

- Developer group G_1 is working on project P. Developer group G_2 wants to make private modifications to P for its own purpose, and continue tracking changes G_1 makes to P.

- Developer group G_1 is working on project P. Developer group G_2 wants to make modifications to P, and propagate those modifications back to G_1, but it doesn't want to use the same CVS server infrastructure as G_1.

- Developer group G is working on project P. It wants to maintain an internal CVS server and a public CVS server; the private server is used for development, and the public CVS server is used to provide the user and the developer community with read-only access to selected versions of the source tree.

- Developer group G is working on project P. It wants to maintain a repository with every revision of every source file in a project. It also wants to maintain a separate repository containing only selected versions of source files in the project (for example, only publicly released versions). The second repository would be considerably smaller and could be used for various archival purposes.

- A developer going off-line for an extended period of time could create a local repository on his private computer, and propagate his changes back to the project repository after returning on-line.

Chapter 2

Existing CVS repository model

In order to understand the changes necessary to the existing CVS architecture, some of its key properties need to be specified.

CVS client-server architecture can be thought of as consisting of four parts:

- the repository, stored on a server;
- the client-server protocol, used to communicate between a client and a server;
- the working files, stored on a client; and
- the user interface, used to communicate between a client and a user, or a server and a user (for administration).

2.1 Repository structure

A repository is a hierarchy of files residing on the server. Every file has a *history*, which encompasses all revisions of that file since its creation.

The history of each file is also hierarchical. The revision tree of a file consists of *branches*, every branch containing an arbitrary number of revisions. Conceptually, a branch corresponds to a line of development in the life of a project; for example, concurrent development of two separate versions might use two branches to maintain separate histories of the two versions.

Every revision in the tree is uniquely identified with a *revision number*, which is an ordered n-tuple of integers. The last component of a revision number identifies the revision on a branch, and the remaining components identify the branch. Revision numbers are usually written as dot-separated strings. For example, the n-tuple $(1, 2, 3, 4)$ would be written as `1.2.3.4`, and would represent revision `4` on branch `1.2.3`.

Revisions are numbered sequentially, so revision `x.2` would immediately precede revision `x.3` and immediately follow revision `x.1`. The first revision of a file has the number `1.1`. The initial branch of a file (with branch number `1`) is called *the trunk*.

Every branch has a *head revision*, which is the latest revision on the branch. Every branch except for the trunk has a *branch point*, which is the revision from which is was derived. The branch point of a branch is a revision on a different branch – its *parent branch*. The branch number of every branch consists of the branch point revision number, followed by its *branch index*, which is an even integer [1]. Branch indices are assigned sequentially, startying at 2. For example, the branch with number `1.2.4.6.8` would be the fourth branch whose branch point is at the revision `1.2.4.6` (which in turn would be the sixth revision on the second branch branched from revision `1.2`).

Every revision and branch can carry any number of *symbolic tags*, which are character strings. [2] No two revisions or branches of a single file can carry the same symbolic tag. Symbolic tags are used to label specific revisions to identify them by their function in the project development. For example, *"Final_3_0_version"* might be a tag assigned to the revisions of all the files in a repository which were part of the final 3.0 version of the product. This allows them to be easily identified later, since individual revision numbers of different files are likely to be significantly different.

[1] The only exceptions are the vendor branch, used when importing vendor sources into a repository, whose branch number is 1.1.1, and the trunk, whose branch number is 1.

[2] Only alphanumberic characters, underscores, and hyphens are allowed in symbolic tags

2.2 Client-server protocol

The first step in the client-server protocol is authentication. CVS supports three different authentication mechanisms: password, Kerberos, and GSSAPI. The server distinguishes between them based on which network port received the authentication request.

After authentication completes successfully, the client sends commands to the server. Every command consists of a command name (such as *"checkout"*), options (whose allowed values depend on the command), and arguments (usually names of the files the command is to operate on). The server sends a response for every command it receives. Some commands require the client and the server to exchange contents of files. If a file is sent from the client to the server, it is sent before any commands that affect it, and then referred to in the commands. If a file is sent from the server to the client, it is sent as a part of a response.

The client-server protocol shows the roots of CVS. The initial version of CVS required the repository and the working files to be available to the CVS program via the file system. With the addition of the client-server model, the client sends the server enough information to recreate a portion of the client's file system in a temporary directory on the server. The server then executes the specified commands as if it were not running in the client-server mode, and returns the responses to the server.

2.3 Working files

Working files reside in the client's file system. For each file present among the working files (not all files from the repository have to be present), the client stores the contents of one revision of the file, possibly with local modifications .

For each directory containing working files, the CVS client maintains information needed to communicate with the server in a subdirectory named *"CVS"*, also known

as the *administrative directory*. The contents of an administrative directory, known as the *administrative files*, contain

- the location of the repository from which the directory was retrieved;
- the location of the directory within the repository; and
- a list of files in the directory, together with per-file bookkeeping information such as the revision number of the revision which was last retrieved from the repository.

2.4 User interface

The user interface, in both the command-line versions of the CVS client (UNIX, Windows), and graphical versions of client (Macintosh, Windows), allows the user to specify a number of files in the working directory and a command to perform on them. Where applicable, there might be optional parameters to refine the command.

The client then translates the command to a request the server understands. Since the server essentially executes the commands as if they had been issued locally on the server, most commands require very little translation.

Chapter 3

Modifications to the repository model

The fundamental property of a distributed repository is that it is possible for different parts of the history of a file to be on different servers; at the same time, the client still has to be able to present them as a single entity to the user.

Starting from the existing repository model, the main new concept necessary to support a distributed repository is that a server might not be able to fullfill client's request, because it needs revision information carried by a different server. However, if the repository is allowed to be distributed arbitrarily, then complexity of any operation on the repository grows beyond practical limits.

To reduce complexity, we need to take advantage of the fact that the full generality of a distributed system is not needed to support objectives outlined in the first chapter. In addition to that, we note the following fact about the way people typically use CVS:

Actions which operate on two or more revisions of a file which are on different branches are rare. The majority of operations only use one or more revisions on one branch.

We can use this observation to form the basic premise of a distributed CVS system which fulfills the described without the maximum possible generality:

All revisions on a particular branch of the revision tree of a file must reside on a single server.

With that restriction, a server will have all the necessary information for most CVS operations. Therefore, most operations in a distributed repository will behave exactly the same as they would in a non-distributed repository.

This restriction simplifies the problem considerably; it becomes unnecessary to consider arbitrary arrangements of servers and arbitrary distributions of revisions among servers. With this restriction, the performance of the client and server are essentially unaffected for vast majority of CVS operations, simply because the majority of CVS operations take place on a single branch. Also, the amount of additional state stored on the server is reduced, because only branches incur additional state, whereas there is no additional state needed for each revision.

Since revisions and branches are arranged in a tree, and a transition from one server to another can only occur at a branch point, servers are effectively also arranged in a tree. Because of that, this is a *hierarchically distributed repository*.

To implement hierarchical repositories, modifications are necessary to each of the four subdivisions of CVS presented in the previous chapter. Overall, the changes introduce the idea that a server might not know a particular revision's contents or even whether it exists, but instead knows which other server might have that information.

3.1 Changes to repository structure

The existing repository structure already encompasses information about branches and branch points. In order to support the notion of a remote branch (i.e. a branch which exists on a different server), the repository structure has to be extended with two pieces of information:

- Each revision has to include locations of all branches at the revision which are not carried by the server
- A branch whose parent branch is carried by a different server has to include the location of the parent branch.

The RCS file format, used by the server for storage of all revision information, is very flexible. It is essentially a tagged data format; it consists of a list of records

corresponding to individual revisions, each record listing a number of fields in the form of key-value pairs. The fields describe properties of each revision; therefore, it is not difficult to add additional fields to list the locations and the branch numbers of remote branches at a branchpoint, and the location of the parent branch.

For each remote branch, in addition to its branch number, the RCS file has to include the location where the branch can be found. This location is a repository root, as already used in CVS, without authentication information (since it is the client, not the server, that determines the authentication protocol to be used).

3.2 Changes to client-server protocol

Some commands issued by the CVS client don't operate on any files. Those commands require no modification, as it is only per-file information which is different in the hierarchical repository model.

The remaining commands operate on one or more files; whenever they operate on more than one file, the operations on different files are independent of each other. Therefore, the commands could just as well be operating on single files, and the ability to operate on multiple files is included mainly for user convenience. Because of that, we can consider the commands as if they operate on single files.

Of the commands that operate on files, some operate on a single revision of a file, some operate on two revisions of a file, and some operate on three revisions of a file. Although some commands' parameters determine whether the command operates on one, two, or three revisions of a file, those cases can be considered as separate commands, since the combination of command and parameters always uniquely determines how many revisions the command operates on.

Every command operating on a single revision must be modified to accept a new response, by which the server indicates to the client that the requested revision is not carried by that server, and refers the client to a different server. The client can then retry the operation on the other server (which might again redirect the client).

Commands operating on two or three revisions have the potential difficulty that some of those revisions might be carried by different servers. In that case, the client must be able to locate all but one of the revisions on different servers (using the same procedure as it would for single-revision commands), and then pass that information to the server carrying the last one. The server then has the information about all of the revisions, and can complete the command. Putting the burden of locating all the necessary information on the client reduces the impact of distributed model on server performance.

The `tag` command, used to create branches (and other symbolic tags), has to be modified to allow creation of remote branches. Remote branch creation has to be performed in two steps: first the server carrying the parent branch is contacted to determine the branch number of the new branch, then the server which will carry the new branch is contacted to create the branch. The parent server has to return information to the client about the new branch, which the client then transmits to the child server. This requires modifying the protocol to allow the `tag` command to return additional information, and to allow the remote branch to be created with this additional information.

Authentication in the existing CVS protocol is inadequate for distributed repositories, because there are no provisions for authentication negotiation between the client and the server. Since it is possible that a client might need to contact several servers to complete an operation, the lack of authentication negotation means that a client would have to store authentication information for every server. Instead of imposing that restriction, the authentication exchange will be modified to include negotiation of the authentication mechanism; then the client can hop from one server to another without requiring each one of them to be configured on the client. This will work especially well with Kerberos or GSSAPI authentication, as the client can present user's credentials without prompting the user for a different password for every server.

3.3 Changes to working files

The existing structure of working files stores information per-directory. All the files in a single directory are assumed to come from the same server. However, in a distributed repository, files within one directory might come from different servers. Therefore, this restriction has to be relaxed to allow additional per-file administrative information.

3.4 Changes to user interface

Since remote repositories correspond to branches in the revision tree, the existing facilities for creating a branch can be extended to allow creating remote branches. The `tag` command creates a new branch of a file, branching off the revision of the file currently in the working files.

To allow creating remote branches, the `tag` command can be extended to allow an argument specifying the repository on which the branch should be created, as follows:

Create a standard branch: `cvs tag -r BranchName`

Create a remote branch: `cvs tag -r :Server:Repository:BranchName`

This syntax also follows the established CVS syntax for specifying a repository location by separating the components with a ':'. The branch is created on the specified repository, which must be created separately, using `cvs init`. Repository creation and branch creation are separate because creating a repository is usually a task which only a small group of trusted people is allowed to perform, and therefore allowing a repository to be implicitly created with the `tag` command would be inappropriate.

Chapter 4

Implementation

4.1 Repository structure

Every file in the repository consists of a header, which contains information common to all revisions of the file, and a list of revisions.

The header contains the version number of the head revision (the most recent revision on the main branch) and a list of all symbolic tags.

Every revision contains a list of branches at that revision and the revision which immediately precedes it on the branch (or none, if that is the first revision on the branch).

Every field in the revision file has the form `<key> [<value>] [<value>] [...];`. The values are optional; the values are separated from the key and from each other with whitespace.

The list of remote branches at a revision is stored in a new `remote-branches` field. The format of each value in the list of remote branches is `<revision><location>`, where the revision is the branch number of the remote branch, and the location is its location. The location is given as a CVS root – `:method:server:repository`. The method field is unused in the current implementation, but is included to avoid introducting a new way to specify CVS repositories.

The location of the parent branch is stored in a new `parent-branch` field in the RCS file header. This field is also given as a CVS root.

The RCS subsystem of CVS has to be modified to parse the new fields of in the RCS files. The list of branchs at a revision was added to the `rcsversnode` structure, where as the parent branch location was added to the `rcsnode` structure. The `RCS_parse()` function was modified to parse the additional fields and insert them into the structures as appropriate.

4.2 User interface

The only change in the user interface is the ability to specify remote branches for the `tag` command. The only change to the corresponding code was to allow the `tag()` function to recognize remote tags as valid.

However, there are several related changes to the client-server protocol, which are described in the next section.

4.3 Client-server protocol

4.3.1 Authentication exchange

As metioned previously, the existing authentication exchange is inadequate, because it requires the client to know the authentication method to be used for a particular server before establishing connection with that server (different authentication methods are served on different ports).

In the existing CVS protocol, after connection is established (and authentication method is therefore known), the client authentication request consist of `"BEGIN AUTHENTICATION REQUEST"` followed by the username, the authentication data, and `"END AUTHENTICATION REQUEST"`. This concludes the authentication request; the server responds with `"I LOVE YOU"` or `"I HATE YOU"`.

In the modified authentication exchange, the server begins by transmitting a space-separated list of allowed authentication methods. For example, the server might transmit `"KERBEROS_V4 GSSAPI PASSWORD"`.

The client then picks one of the authentication methods, and transmits "BEGIN *authentication-method* AUTHENTICATION REQUEST".

Following that, the client sends the username on a line by itself, followed by the repository root on a line by itself.

Sending the repository root during the authentication exchange allows the server to perform per-repository authentication upfront, rathen than allowing the client to access the server at first, but then denying access later (when the Root command is transmitted).

After the repository root, the client sends authentication data, which varies depending on the authentication method. For password authentication, it's the encrypted password; for Kerberos, it's the authenticator.

Finally, the client sends "END AUTHENTICATION REQUEST", thus concuding the authentication request. The server responds with "I LOVE YOU" or "I HATE YOU", as before.

4.3.2 Server responses

Not-carried

One new server response is required in the protocol to support distributed repositories. Any command which cannot be completed because its completion requires revisions which are not present on the server generates the Not-Carried response.

The Not-Carried response has the following form:

Not-Carried revision path –server –repository –directory –Not-Carried
revision path –server –repository –directory –... –End-Not-Carried

The response consists of any number of four-line blocks specifying the file and revision which are not carried by the server, and where the client should proceed to look for them. The blocks are followed by End-Not-Carried to indicate the end of the response.

Most commands only generate a Not-Carried response for only one revision, since most commands operate on a single revision of a file. However, some commands can

return `Not-Carried` responses with more than one revision – for example, the update command.

This form of the response allows a client to associate all parts of the response with the command it had issued, but also allows the client to treat multiple revisions in the response as separate entities. The client's implementation might make one interpretation or the other more appropriate. For example, a client implementation which maintains association between requests and responses (common in clients with graphical user interface) would want to know whether two `Not-Carried` responses are generated in response to the same request or not. On the other hand, a client implementation which handles all responses after sending all requests (common in command-line clients) might not care whether two `Not-Carried` responses are related or not.

For each revision specified in the response, the following information is included:

- revision: the revision number (or tag) to which this information pertains. This allows the client to determine what to ask of the other server.
- path: the client's path to the file to which this information pertains. This is the same as the path which the client passed to the server, and is included so that clients do not have to carry such context information from the time that the request is made to the time when the response is received. [1]
- server: the host name of the other carrying the specified revision. The name should be a fully qualified name.
- repository: the location of the repository on the other server.
- directory: the location of the file in the repository on the other server; in general, this is a subdirectory of the repository path.

With that information, the client can proceed to the other server, which might be able to complete the request, or might issue more `Not-Carried` responses, redirecting

[1] One might argue that a client should carry such information in order to provide better error reporting and user interaction, but the most common CVS client does not do that, and it is by far simpler to include this additional piece of information in the response than it would be to rearchitect the client.

the client again. In either case, The `Not-Carried` responses provide the client with enough information to find the needed revisions and complete the desired operation.

The CVS server implementation uses the `Classify_File()` function to determine the state of a file in the repository. Therefore, introducing the "Not carried" status involved modifying `Classify_File()` to add the logic to detect when a revision might exist on a different server.

`Classify_File()` in turn uses `Version_TS()` to retrieve RCS information about a file, so `Version_TS()` also had to be modified to handle remote revisions.

`Version_TS()` internally used `RCS_getversion()` to establish which RCS version corresponds to a CVS revision or symbolic tag. Since many other places in CVS use `RCS_getversion()`, it was not modified. A new function, `RCS_getremoteversion()` was added instead; it performs the same function as `RCS_getversion()`, but only handles remote revisions.

Once `Classify_File()` was modified to return the "not carried" status, the places which call `Classify_File()` had to be modified to handle the new status.

The client handles `Not-carried` responses in the `handle_not_carried()` function, which is added to the client's response handler table. Contents of the `Not-carried` response are added to a queue by `handle_not_carried()`. After all the responses from the server are received, processing turns to the queue and handles all outstanding `Not-carried` responses.

In most cases, handling a queued `Not-carried` response involves repeating the same command on a different server. The only exception are commands which operate on more than one revision of a file: difference and update. For those commands, handling a queued `Not-carried` response involves retrieving the remote revisions from a different server and then returning to the first server to complete the command.

Queue processing is handled by the `client_process_remotes()` function, which traverses the queue and performs the appropriate action on each entry.

If the action is to repeat the command on a different server, a new argument list is assembled by the `setup_args()` function, and a new CVS root is configured by the

`setup_root()` function, after which the CVS command parsing is invoked to execute the command as if it had been invoked directly by the user.

If the action is to fetch a revision from a remote server, a new argument list and a new root are built, but rather than invoking the original command again, the `update` command processor is invoked, with arguments specifying that the revision should be directed to the standard output stream.

After the remote revisions are retrieved, queue processing returns to processing the original command, by reissuing the command to the first server, but also including the contents of the remote revisions, as described in the next section.

`Create-remote-branch`

When a server is asked to create a remote branch in the revision tree (using the modified `tag` request described below), it examines the existing revision tree and determines the branch number of the new branch. It reserves this branch number in the revision tree of the file, and then returns the revision number of the new branch in a `Create-remote-branch` response, which takes the following form:

`Create-remote-branch` filename –server –repository –path –revision

The response includes the name of the file, the full location of the new remote branch (the name of the server, the location of the repository, and the location of the file in the repository), followed by the branch number of the new branch.

The client handles this new response by enqueuing a new request to be sent to the server where the remote branch will reside. This request will create the remote branch itself, as described below.

4.3.3 Client requests

The syntaxes of most existing client requests are unchanged in the distributed repository model. However, when different revisions needed to complete a request reside on different servers, the client has to have a mechanism of transferring this information

from one server to another. In addition to that, the process of remote branch creation requires additional information in some existing requests.

`Remote-revision`

Since CVS already has a mechanism for retrieving arbitrary revisions of a file, it is easy for the client to retrieve a revision from one server in order to transfer it to another server. The existing `update` request is used for that, with the only difference being that the file returned by the server is saved in the administrative `CVS` directory, rather than in the working directory.

However, in order to send the contents of a revision to the server, without having the server change the contents of the repository, the client needs a new request, `Remote-revision`. With this request, the client sends the contents of a remote revision to the server, and the server retains that information until the end of the session. During the session, the server can use the information to complete requests which might otherwise elicit a `Not-carried` response.

In fact, the format of the `Remote-revision` request does not need to differ significantly from the existing `Modified` request, which sends the contents of a client's working file:

`Remote-revision` filename –revision –mode –size –contents

The request includes the name of the file provided in the request, followed by the revision number, the file mode (as defined by the file system), the size (in bytes), and by the contents of the revision. Note that no RCS keyword substitution is performed on the contents of the file.

The CVS client implementation sends information about every working file affected by a command by recursively traversing the working files and directories, and sending the status and, if necessary, the contents of the files. This is done from the `send_fileproc()` function, which determines the status of a file, and sends appropriate information to the server.

`Remote-revision` is added sent to the server by `send_fileproc()`, along with the other information about working files. `send_fileproc` calls a new function, `send_remoterev()`, which sends the contents of a remote revision, as received while handling a `Not-carried` response.

On the server, the `Remote-revision` request is handled by saving the contents of the remote revision in a temporary file in the new `serve_remote_revision` function. Additionally, the logic for determining whether a client command will result in a `Not-carried` response has to be extended to avoid returning that response when the remote revision has already been provided by the client. This is accomplished by making further modifications to `Classify_File()`.

In order to allow the RCS subsystem to be able to perform operations on the remote revisions, the remote revisions have to be temporarily injected into the RCS revision tree of the file. When `Classify_File()` is asked to classify a remote revision which has been provided by the client in a `Remote-revision` request, it inserts the contents of the revision into the revision tree on the special branch 1.1.3, and classifies the revision as local. At the same time, it changes the real remote revision number to the revision on the 1.1.3 branch. Therefore, any calls to `Classify_File` are automatically redirected to use the local data for remote revisions.

`tag`

The `tag` request is used to create symbolic revisions and branches. It directly corresponds to the `tag` command in the user interface. Therefore, augmentic the `tag` command to accept the new syntax for remote branch creation directly affects the `tag` request, which now also has to handle remote branch creation.

The branch name argument, which in the `tag` command is now allowed to be the name of a remote branch, is passed directly to the server in a `tag` request. Other than allowing the new (previously invalid) values for this argument, the request is unchanged.

Server processing of this request is different. In addition to finding a new branch number and creating the branch, the server has to return a `Create-remote-branch`

response to the client, instructing the client to contact the server on which the branch will be carried. To accomplish this, first the function `tag_fileproc()` had to be modified to recognize remote branch names. When a remote branch name is detected by `tag_fileproc()`, it calls a new function, `RCS_setremotetag()`, which functions similarly to `RCS_settag()`: it finds a free branch number at the desired revision, and reserves it in the revision tree, updating the file in the repository.

After `RCS_setremotetag()` returns the new remote branch revision, the information is returned to the client by `tag_fileproc()` in a `Create-remote-branch` response, described in the previous section.

`add`

When the client receives the `Create-remote-branch` response, it handles it by establishing a connection with the server which will carry the remote branch. This server has to create a new file in the repository, setting its parent branch to the one specified by the client.

To transmit this information from the server to the client, a new argument was added to the `add` request. In non-distributed repositories, `add` tells the server a file is about to be added to the repository. This request is augmented with a new argument, which specifies the location of the parent branch. The new argument is of the form

```
-b :server :repository :revision
```

The server and the repository strings specify the location of the parent branch, whereas the revision is the revision number of the new branch.

In response to this request, the server creates an empty revision file in the repository, only setting its parent branch from the information in the request. New revisions committed to this file are committed to the specified branch.

The function `add_rcs_file()` is used to create the new empty revision file. It was already used by `commit()` and `import()` to create new files in the repository. It had to be extended to take a new argument specifying the remote branch name. It is then called by `add()` in respose to a request to add a new remote branch.

Chapter 5

Discussion

5.1 Implementation analysis

At every step from the initial design througout the implementation, the project was constrained by being implemented as an extension to CVS rather than a replacement for CVS.

A revision control system typically maintains many man-years of project history, and therefore development teams are understandably reluctant to make any modifications to their existing version control system, especially if those changes break backwards compatibility with the existing repositories or existing clients. Other things being equal, acceptance of a revision control system is chiefly determined by how difficult it is to switch to it from existing version control systems.

In addition to that, the focus of this thesis is on distributed repositories, and therefore I considered it undesirable to write a complete revision control system merely to serve as the basis for my extensions.

Hence, the intent was to demonstrate that the problems described in Chapter 1 can be solved by extending an existing widely used revision control system. On one hand, this was a tremendeous benefit, because many basic problems of revision control had already been solved. File merging, histories, branches, tags, and many other things were already in place.

On the other hand, during the implementation of my thesis, I discovered that the implementation I used as a starting point was not at all designed with future extensions in mind. This turned out to be a major hindrance in my implementation, and is the main reason why some important work had to be left out of my implementation.

The main difficulty with the implementation I used is that it had been worked on by several different people and teams at different times, but there was very little communication among them.

The resulting design is somewhat incoherent and not easily extended. In its current state it seems to be factored and separated into manageable parts, but on a lower level each of those parts is a combination of several incompatible intents, all pulling in different directions. The structure of the code has reached the point where changing one thing can cause semingly unrelated others to fail. This was a major problem to me, and unfortunately it did not become obvious until it was too late to consider different starting points or even writing everything from scratch.

Regrettably, but inevitably, the changes I made only increased the strain in the code.

My final implementation works sufficiently to demonstrate that the idea of hierarchical repositories can solve the problems which it was intended to solve, and can be implemented as an extension to CVS. However, the implementation is far from robust and not something I would reccomend for a production environment. Many of its aspects are untested or unpolished.

However, I do believe that hierarchically distributed repositories are feasible and useful.

Recently, the computing industry has seen a tremendous increase in visibility of open-source software, and in interaction between commercial software and open-source software. Any project faces the possibility of being partly proprietary and partly open-source, either as an open-source project being extended by a commercial vendor, or as a commercial product becoming open-source. This implies that different parts of one project might be managed by different people, teams, or even companies,

and it is therefore paramount that the version control system gives them the ability to decentralize the project to parallel the decentralization of control.

Thus, my recommendation to those interested in distributed version control is:

- If you are designing a new version control system, consider distributed repositories in your design, even if you do not intend to implement them initially. Allowing for that possibility early on will likely greatly reduce the effort needed to implement them when the market demands it.
- If you are extending a new version control system to use distributed repositories, consider whether the implementation you start from is extensible; unfortunately, this is not necesarily easy to determine without trying to extend it.
- If you are extending CVS to use distributed repositories, consider rewriting the rest of CVS before you start, or using a different starting implementation from the one I used. As of this writing, there aren't any other implementations of both the client and the server used, but there is at least one free, independent, and much better thought out implementation [4] of the client which, in hindsight, might have been a more fruitful starting point.

5.2 Further work

The `update` command has not been extended to allow merging of revisions from different servers. However, most of the functionality necessary to support `update` was also necessary to support `diff` and therefore only a comparably small amount of additional work would be required to fully support `update`.

The minor problems of which I am aware, but which I have not had the time to resolve are:

- Modify the client to coallesce requests for multiple remote revisions from one server into a single request. For example, if three remote revisions are require to complete a command, and two of them reside on the same server, them both can be retrieved during a single connection to the server, rather than being retrieved with two separate connections to the server. My implementation keeps the out-

standing requests for remote revisions in a souble-ended queue, and processes the queue in order. Coallescing the requests could be done by inserting additional outstanding remote revision requests immediately before or after existing requests, rather than at the beginning or the end of the queue.

- Eliminate remaining resource leaks from the client. There are several resource leaks in the client, primarily in constructing argument lists for commands executed in response to `Not-carried` server responses. These leaks can become significant if there are many servers in a distributed repository, because the expected number of Not-Carried responses grows with the size of the server hierarchy.
- Improve error handling in handling of remote revisions. Like the rest of CVS, the code which handles remote revisions aborts client execution when encountering a fatal error. However, some errors which are currently considered fatal don't need to be fatal if appropriate status information can be returned to the higher-level code, and handled there. Unfortunately, error reporting capabilities in the underlying implementation are somewhat primitive, and therefore improving error handling might require modifications to the error-handling code.
- Add client-side caching of the repository hierarchy. In handling a request for a remote revision, much time is spent determining where the revision is, compared to the amount of time spent handling the request once the revision is found. The time spent discovering the repository hierarchy grows with the size of the hierarchy, and therefore reducing that time can significantly improve client performance in large hierarchies. Since it is impossible for more than one server to carry a particular revision, it is easy to verify whether cache entries are valid or stale simply by attempting to retrieve the revision from the server.
- Improve performance of the authentication exchange. Currently, there is a noticeable delay during the authentication exchange, which hasn't been fully diagnosed. It seems as if the client and the server spend a brief period of time waiting on each other, when they could just proceed with the rest of the exchange. The delay becomes significant in larger server hierarchies because of the need to authenticate to multiple hosts in order to complete the command.

- Submit the changes back to the CVS maintainers. In the spirit of open source software, the modifications made in this thesis will be submitted back to the maintainers so they can be considered for inclusion in the official release of CVS.

Chapter 6

Conclusion

Distributed repositories can be applied to several important scenarios which occur in software development, especially with the recent surge in interaction between commercial and open-source software. They allow the revision control of a project to be distributed in way which parallels the distribution of project management.

The CVS protocol is suitable for an extension which implements distributed repositories with minimal schanges to the protocol; unfortunately, because of inadequacies in the authentication exchange of the CVS protocol, it is infeasible to implement hierarchical repositories in a completely backwards-compatible way. A client which does not understand distributed repositories cannot communicate with a server which only understands distributed repositories; however, it is possible to create a server which understands both (and therefore give client with no knowledge of distributed repositories access to individual servers in a distributed repository) simply by using a different network port for clients which are aware of distributed repositories.

The existing CVS implementation by Cyclic, version 1.10 or older, is unsuitable for significant extensions, and should be avoided as a starting point. Other implementations, such as the one which forms the core of MacCVS Pro, a Mac OS CVS client, might be better suited for such extensions.

The ideas and intents of distributed repositories apply to other version control systems, and I urge the designers of new version control systems (of which there are

several as of this writing, partly due to the state of CVS), to design their systems with distributed repositories in mind.

Appendix A

Source code

This chapter contains the entire source code for the CVS client and server with support for hierarchically distributed repositories. The differences between the version of CVS used as starting point (the Cyclic version 1.10) and the final version with support for distributed repositories are marked with change bars.

A.1 add.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Add
 *
10  * Adds a file or directory to the RCS source repository.  For a file,
 * the entry is marked as "needing to be added" in the user's own CVS
 * directory, and really added to the repository when it is committed.
 * For a directory, it is added at the appropriate place in the source
 * repository and a CVS directory is generated within the directory.
 *
 * The -m option is currently the only supported option.  Some may wish to
 * supply standard "rcs" options here, but I've found that this causes more
 * trouble than anything else.
 *
20  * The user files or directories must already exist.  For a directory, it must
 * not already have a CVS file in it.
 *
 * An "add" on a file that has been "remove"d but not committed will cause the
 * file to be resurrected.
 */

#include "cvs.h"
#include "savecwd.h"
#include "fileattr.h"
30

static int add_directory PROTO ((struct file_info *finfo));
static int build_entry PROTO((char *repository, char *user, char *options,
                              char *message, List * entries, char *tag));

static const char *const add_usage[] =
{
    "Usage: %s %s [-k rcs-kflag] [-m message] files...\n",
    "\t-k\tUse \"rcs-kflag\" to add the file with the specified kflag.\n",
    "\t-m\tUse \"message\" for the creation log.\n",
40  "(Specify the --help global option for a list of other help options)\n",
```

```
    NULL
};

int
add (argc, argv)
    int argc;
    char **argv;
{
    char *message = NULL;
    char *revision = NULL;
    int i;
    char *repository;
    int c;
    int err = 0;
    int added_files = 0;
    char *options = NULL;
    List *entries;
    Vers_TS *vers;
    struct saved_cwd cwd;
    /* Nonzero if we found a slash, and are thus adding files in a
       subdirectory.  */
    int found_slash = 0;

    if (argc == 1 || argc == -1)
        usage (add_usage);

    wrap_setup ();

    /* parse args */
    optind = 0;
    while ((c = getopt (argc, argv, "+k:m:r:")) != -1)
    {
        switch (c)
        {
            case 'k':
                if (options)
                    free (options);
                options = RCS_check_kflag (optarg);
                break;

            case 'm':
                message = xstrdup (optarg);
                break;

            case 'r':
              if (adding_remote) {
                revision = xstrdup (optarg);
              } else {
                error (1, 1, "-r is not valid as a user option to add");
              }
              break;

            case '?':
            default:
                usage (add_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    if (argc <= 0)
        usage (add_usage);

    /* First some sanity checks.  I know that the CVS case is (sort of)
       also handled by add_directory, but we need to check here so the
       client won't get all confused in send_file_names.  */
    for (i = 0; i < argc; i++)
    {
        int skip_file = 0;

        /* If it were up to me I'd probably make this a fatal error.
           But some people are really fond of their "cvs add *", and
           don't seem to object to the warnings.
           Whatever.  */
        strip_trailing_slashes (argv[i]);
        if (strcmp (argv[i], ".") == 0
            || strcmp (argv[i], "..") == 0
            || fncmp (argv[i], CVSADM) == 0)
        {
            error (0, 0, "cannot add special file `%s'; skipping", argv[i]);
            skip_file = 1;
        }
        else
        {
            char *p;
            p = argv[i];
            while (*p != '\0')
            {
                if (ISDIRSEP (*p))
```

```
            {
                found_slash = 1;
                break;
            }
            ++p;
        }
    }

    if (skip_file)
    {
        int j;

        /* FIXME: We don't do anything about free'ing argv[i].  But
           the problem is that it is only sometimes allocated (see
           cvsrc.c).  */

        for (j = i; j < argc - 1; ++j)
            argv[j] = argv[j + 1];
        --argc;
        /* Check the new argv[i] again.  */
        --i;
        ++err;
    }
}

#ifdef CLIENT_SUPPORT
if (client_active)
{
    int i;

    if (argc == 0)
        /* We snipped out all the arguments in the above sanity
           check.  We can just forget the whole thing (and we
           better, because if we fired up the server and passed it
           nothing, it would spit back a usage message).  */
        return err;

    start_server ();
    ign_setup ();
    if (options) send_arg(options);
    option_with_arg ("-m", message);

    /* If !found_slash, refrain from sending "Directory", for
       CVS 1.9 compatibility.  If we only tried to deal with servers
       which are at least CVS 1.9.26 or so, we wouldn't have to
       special-case this.  */
    if (found_slash)
    {
        repository = Name_Repository (NULL, NULL);
        send_a_repository ("", repository, "");
        free (repository);
    }

    for (i = 0; i < argc; ++i)
        /* FIXME: Does this erroneously call Create_Admin in error
           conditions which are only detected once the server gets its
           hands on things?  */
        if (isdir (argv[i]))
        {
            char *tag;
            char *date;
            int nonbranch;
            char *rcsdir;
            char *p;
            char *update_dir;
            /* This is some mungeable storage into which we can point
               with p and/or update_dir.  */
            char *filedir;

            if (save_cwd (&cwd))
                error_exit ();

            filedir = xstrdup (argv[i]);
            p = last_component (filedir);
            if (p == filedir)
            {
                update_dir = "";
            }
            else
            {
                p[-1] = '\0';
                update_dir = filedir;
                if (CVS_CHDIR (update_dir) < 0)
                    error (1, errno,
                           "could not chdir to %s", update_dir);
            }

            /* find the repository associated with our current dir */
            repository = Name_Repository (NULL, update_dir);
```

```
                /* before we do anything else, see if we have any
                   per-directory tags */
                ParseTag (&tag, &date, &nonbranch);

                rcsdir = xmalloc (strlen (repository) + strlen (p) + 5);
                sprintf (rcsdir, "%s/%s", repository, p);

                Create_Admin (p, argv[i], rcsdir, tag, date,
                              nonbranch, 0);

                if (found_slash)
                    send_a_repository ("", repository, update_dir);

                if (restore_cwd (&cwd, NULL))
                    error_exit ();
                free_cwd (&cwd);

                if (tag)
                    free (tag);
                if (date)
                    free (date);
                free (rcsdir);

                if (p == filedir)
                    Subdir_Register ((List *) NULL, (char *) NULL, argv[i]);
                else
                {
                    Subdir_Register ((List *) NULL, update_dir, p);
                }
                free (repository);
                free (filedir);
            }
        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, 0, 0, SEND_BUILD_DIRS | SEND_NO_CONTENTS);
        send_to_server ("add\012", 0);
        if (message)
            free (message);
        return err + get_responses_and_close ();
    }
#endif

    /* walk the arg list adding files/dirs */
    for (i = 0; i < argc; i++)
    {
        int begin_err = err;
#ifdef SERVER_SUPPORT
        int begin_added_files = added_files;
#endif
        struct file_info finfo;
        char *p;

        memset (&finfo, 0, sizeof finfo);

        if (save_cwd (&cwd))
            error_exit ();

        finfo.fullname = xstrdup (argv[i]);
        p = last_component (argv[i]);
        if (p == argv[i])
        {
            finfo.update_dir = "";
            finfo.file = p;
        }
        else
        {
            p[-1] = '\0';
            finfo.update_dir = argv[i];
            finfo.file = p;
            if (CVS_CHDIR (finfo.update_dir) < 0)
                error (1, errno, "could not chdir to %s", finfo.update_dir);
        }

        /* Add wrappers for this directory.  They exist only until
           the next call to wrap_add_file.  */
        wrap_add_file (CVSDOTWRAPPER, 1);

        finfo.rcs = NULL;

        /* Find the repository associated with our current dir.  */
        repository = Name_Repository (NULL, finfo.update_dir);

        entries = Entries_Open (0, NULL);

        finfo.repository = repository;
        finfo.entries = entries;

        /* We pass force_tag_match as 1.  If the directory has a
           sticky branch tag, and there is already an RCS file which
           does not have that tag, then the head revision is
           meaningless to us.  */
```

```
    vers = Version_TS (&finfo, options, NULL, NULL, 1, 0);
    if (vers->vn_user == NULL)
    {
        /* No entry available, ts_rcs is invalid */
        if (vers->vn_rcs == NULL)
        {
            /* There is no RCS file either */
            if (vers->ts_user == NULL)
            {
                /* There is no user file either */
                error (0, 0, "nothing known about %s", finfo.fullname);
                err++;
            }
            else if (!isdir (finfo.file)
                     || wrap_name_has (finfo.file, WRAP_TOCVS))
            {
                /*
                 * See if a directory exists in the repository with
                 * the same name.  If so, blow this request off.
                 */
                char *dname = xmalloc (strlen (repository)
                                       + strlen (finfo.file)
                                       + 10);
                (void) sprintf (dname, "%s/%s", repository, finfo.file);
                if (isdir (dname))
                {
                    error (0, 0,
                           "cannot add file `%s' since the directory",
                           finfo.fullname);
                    error (0, 0, "`%s' already exists in the repository",
                           dname);
                    error (1, 0, "illegal filename overlap");
                }
                free (dname);

                if (vers->options == NULL || *vers->options == '\0')
                {
                    /* No options specified on command line (or in
                       rcs file if it existed, e.g. the file exists
                       on another branch).  Check for a value from
                       the wrapper stuff.  */
                    if (wrap_name_has (finfo.file, WRAP_RCSOPTION))
                    {
                        if (vers->options)
                            free (vers->options);
                        vers->options = wrap_rcsoption (finfo.file, 1);
                    }
                }

                if (vers->nonbranch)
                {
                    error (0, 0,
                           "cannot add file on non-branch tag %s",
                           vers->tag);
                    ++err;
                }
                else
                {
                    /* There is a user file, so build the entry for it */
                    if (build_entry (repository, finfo.file, vers->options,
                                     message, entries, vers->tag) != 0)
                        err++;
                    else
                    {
                        added_files++;
                        if (!quiet)
                        {
                          if (vers->tag && !revision) {
                              error (0, 0, "\
scheduling %s `%s' for addition on branch `%s'",
                                     (wrap_name_has (finfo.file,
                                                     WRAP_TOCVS)
                                      ? "wrapper"
                                      : "file"),
                                     finfo.fullname, vers->tag);
                          } else if (!vers->tag) {
                              error (0, 0,
                                     "scheduling %s `%s' for addition",
                                     (wrap_name_has (finfo.file,
                                                     WRAP_TOCVS)
                                      ? "wrapper"
                                      : "file"),
                                     finfo.fullname);
                          } else {
                            /* New case: the user file exists, but we are not _really_
                               adding it - we are adding a new remote branch. In this case,
                               we create a new empty RCS file with the appropriate
                               remote branchpoint */
                            add_rcs_file (NULL, vers->srcfile->path, finfo.fullname, NULL,
                                          NULL, NULL, NULL, 0, NULL, NULL, 0, revision, NULL);
```

```
                        }
                    }
                }
            }
        }
        else if (RCS_isdead (vers->srcfile, vers->vn_rcs))
        {
            if (isdir (finfo.file)
410             && !wrap_name_has (finfo.file, WRAP_TOCVS))
            {
                error (0, 0, "\
the directory `%s' cannot be added because a file of the", finfo.fullname);
                error (1, 0, "\
same name already exists in the repository.");
            }
            else
            {
                if (vers->nonbranch)
420                 {
                    error (0, 0,
                           "cannot add file on non-branch tag %s",
                           vers->tag);
                    ++err;
                }
                else
                {
                    if (vers->tag)
                        error (0, 0, "\
430 file `%s' will be added on branch `%s' from version %s",
                               finfo.fullname, vers->tag, vers->vn_rcs);
                    else
                        /* I'm not sure that mentioning
                           vers->vn_rcs makes any sense here; I
                           can't think of a way to word the
                           message which is not confusing.  */
                        error (0, 0, "\
re-adding file %s (in place of dead revision %s)",
                               finfo.fullname, vers->vn_rcs);
440                     Register (entries, finfo.file, "0", vers->ts_user,
                              NULL,
                              vers->tag, NULL, NULL, CVSroot_directory, finfo.repository);
                    ++added_files;
                }
            }
        }
        else
        {
            /*
450          * There is an RCS file already, so somebody else must've
             * added it
             */
            error (0, 0, "%s added independently by second party",
                   finfo.fullname);
            err++;
        }
    }
    else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '\0')
    {
460
        /*
         * An entry for a new-born file, ts_rcs is dummy, but that is
         * inappropriate here
         */
        error (0, 0, "%s has already been entered", finfo.fullname);
        err++;
    }
    else if (vers->vn_user[0] == '-')
    {
470     /* An entry for a removed file, ts_rcs is invalid */
        if (vers->ts_user == NULL)
        {
            /* There is no user file (as it should be) */
            if (vers->vn_rcs == NULL)
            {

                /*
                 * There is no RCS file, so somebody else must've removed
                 * it from under us
480              */
                error (0, 0, "\
cannot resurrect %s; RCS file removed by second party", finfo.fullname);
                err++;
            }
            else
            {

                /*
                 * There is an RCS file, so remove the "-" from the
490              * version number and restore the file
```

```
                 */
                char *tmp = xmalloc (strlen (finfo.file) + 50);

                (void) strcpy (tmp, vers->vn_user + 1);
                (void) strcpy (vers->vn_user, tmp);
                (void) sprintf (tmp, "Resurrected %s", finfo.file);
                Register (entries, finfo.file, vers->vn_user, tmp,
                          vers->options,
                          vers->tag, vers->date, vers->ts_conflict, CVSroot_directory, finfo.repository);
                free (tmp);

                /* XXX - bugs here; this really resurrect the head */
                /* Note that this depends on the Register above actually
                   having written Entries, or else it won't really
                   check the file out. */
                if (update (2, argv + i - 1) == 0)
                {
                    error (0, 0, "%s, version %s, resurrected",
                           finfo.fullname,
                           vers->vn_user);
                }
                else
                {
                    error (0, 0, "could not resurrect %s", finfo.fullname);
                    err++;
                }
            }
        }
        else
        {
            /* The user file shouldn't be there */
            error (0, 0, "\
%s should be removed and is still there (or is back again)", finfo.fullname);
            err++;
        }
    }
    else
    {
        /* A normal entry, ts_rcs is valid, so it must already be there */
        error (0, 0, "%s already exists, with version number %s",
               finfo.fullname,
               vers->vn_user);
        err++;
    }
    freevers_ts (&vers);

    /* passed all the checks.  Go ahead and add it if its a directory */
    if (begin_err == err
        && isdir (finfo.file)
        && !wrap_name_has (finfo.file, WRAP_TOCVS))
    {
        err += add_directory (&finfo);
    }
    else
    {
#ifdef SERVER_SUPPORT
        if (server_active && begin_added_files != added_files)
            server_checked_in (finfo.file, finfo.update_dir, repository);
#endif
    }
    free (repository);
    Entries_Close (entries);

    if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);

    free (finfo.fullname);
}
if (added_files)
    error (0, 0, "use '%s commit' to add %s permanently",
           program_name,
           (added_files == 1) ? "this file" : "these files");

if (message)
    free (message);

return (err);
}

/*
 * The specified user file is really a directory.  So, let's make sure that
 * it is created in the RCS source repository, and that the user's directory
 * is updated to include a CVS directory.
 *
 * Returns 1 on failure, 0 on success.
 */
static int
add_directory (finfo)
    struct file_info *finfo;
```

```
{
    char *repository = finfo->repository;
    List *entries = finfo->entries;
    char *dir = finfo->file;

    char *rcsdir = NULL;
    struct saved_cwd cwd;
    char *message = NULL;
    char *tag, *date;
    int nonbranch;
    char *attrs;

    if (strchr (dir, '/') != NULL)
    {
        /* "Can't happen".  */
        error (0, 0,
               "directory %s not added; must be a direct sub-directory", dir);
        return (1);
    }
    if (fncmp (dir, CVSADM) == 0)
    {
        error (0, 0, "cannot add a `%s' directory", CVSADM);
        return (1);
    }

    /* before we do anything else, see if we have any per-directory tags */
    ParseTag (&tag, &date, &nonbranch);

    /* Remember the default attributes from this directory, so we can apply
       them to the new directory.  */
    fileattr_startdir (repository);
    attrs = fileattr_getall (NULL);
    fileattr_free ();

    /* now, remember where we were, so we can get back */
    if (save_cwd (&cwd))
        return (1);
    if ( CVS_CHDIR (dir) < 0)
    {
        error (0, errno, "cannot chdir to %s", finfo->fullname);
        return (1);
    }
#ifdef SERVER_SUPPORT
    if (!server_active && isfile (CVSADM))
#else
    if (isfile (CVSADM))
#endif
    {
        error (0, 0, "%s/%s already exists", finfo->fullname, CVSADM);
        goto out;
    }

    rcsdir = xmalloc (strlen (repository) + strlen (dir) + 5);
    sprintf (rcsdir, "%s/%s", repository, dir);
    if (isfile (rcsdir) && !isdir (rcsdir))
    {
        error (0, 0, "%s is not a directory; %s not added", rcsdir,
               finfo->fullname);
        goto out;
    }

    /* setup the log message */
    message = xmalloc (strlen (rcsdir)
                       + 80
                       + (tag == NULL ? 0 : strlen (tag) + 80)
                       + (date == NULL ? 0 : strlen (date) + 80));
    (void) sprintf (message, "Directory %s added to the repository\n", rcsdir);
    if (tag)
    {
        (void) strcat (message, "--> Using per-directory sticky tag `");
        (void) strcat (message, tag);
        (void) strcat (message, "'\n");
    }
    if (date)
    {
        (void) strcat (message, "--> Using per-directory sticky date `");
        (void) strcat (message, date);
        (void) strcat (message, "'\n");
    }

    if (!isdir (rcsdir))
    {
        mode_t omask;
        Node *p;
        List *ulist;
        struct logfile_info *li;

        /* There used to be some code here which would prompt for
           whether to add the directory.  The details of that code had
           bitrotted, but more to the point it can't work
```

```
            client/server, doesn't ask in the right way for GUIs, etc.
            A better way of making it harder to accidentally add
            directories would be to have to add and commit directories
            like for files.  The code was #if 0'd at least since CVS 1.5.  */

        if (!noexec)
        {
            omask = umask (cvsumask);
            if (CVS_MKDIR (rcsdir, 0777) < 0)
680         {
                error (0, errno, "cannot mkdir %s", rcsdir);
                (void) umask (omask);
                goto out;
            }
            (void) umask (omask);
        }

        /* Now set the default file attributes to the ones we inherited
           from the parent directory.  */
690     fileattr_startdir (rcsdir);
        fileattr_setall (NULL, attrs);
        fileattr_write ();
        fileattr_free ();
        if (attrs != NULL)
            free (attrs);

        /*
         * Set up an update list with a single title node for Update_Logfile
         */
700     ulist = getlist ();
        p = getnode ();
        p->type = UPDATE;
        p->delproc = update_delproc;
        p->key = xstrdup ("- New directory");
        li = (struct logfile_info *) xmalloc (sizeof (struct logfile_info));
        li->type = T_TITLE;
        li->tag = xstrdup (tag);
        li->rev_old = li->rev_new = NULL;
        p->data = (char *) li;
710     (void) addnode (ulist, p);
        Update_Logfile (rcsdir, message, (FILE *) NULL, ulist);
        dellist (&ulist);
    }

#ifdef SERVER_SUPPORT
    if (!server_active)
        Create_Admin (".", finfo->fullname, rcsdir, tag, date, nonbranch, 0);
#else
    Create_Admin (".", finfo->fullname, rcsdir, tag, date, nonbranch, 0);
720 #endif
    if (tag)
        free (tag);
    if (date)
        free (date);

    if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);

730 Subdir_Register (entries, (char *) NULL, dir);

    cvs_output (message, 0);

    free (rcsdir);
    free (message);

    return (0);

out:
740 if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);
    if (rcsdir != NULL)
        free (rcsdir);
    return (0);
}

/*
 * Builds an entry for a new file and sets up "CVS/file",[pt] by
750 * interrogating the user.  Returns non-zero on error.
 */
static int
build_entry (repository, user, options, message, entries, tag)
    char *repository;
    char *user;
    char *options;
    char *message;
    List *entries;
    char *tag;
760 {
```

```
    char *fname;
    char *line;
    FILE *fp;

    if (noexec)
        return (0);

    /*
     * The requested log is read directly from the user and stored in the
     * file user,t.  If the "message" argument is set, use it as the
     * initial creation log (which typically describes the file).
     */
    fname = xmalloc (strlen (user) + 80);
    (void) sprintf (fname, "%s/%s%s", CVSADM, user, CVSEXT_LOG);
    fp = open_file (fname, "w+");
    if (message && fputs (message, fp) == EOF)
        error (1, errno, "cannot write to %s", fname);
    if (fclose(fp) == EOF)
        error(1, errno, "cannot close %s", fname);
    free (fname);

    /*
     * Create the entry now, since this allows the user to interrupt us above
     * without needing to clean anything up (well, we could clean up the
     * ,t file, but who cares).
     */
    line = xmalloc (strlen (user) + 20);
    (void) sprintf (line, "Initial %s", user);
    Register (entries, user, "0", line, options, tag, (char *) 0, (char *) 0, CVSroot_directory, repository);
    free (line);
    return (0);
}
```

A.2 admin.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Administration ("cvs admin")
 *
 */

#include "cvs.h"
#ifdef CVS_ADMIN_GROUP
#include <grp.h>
#endif
#include <assert.h>

static Dtype admin_dirproc PROTO ((void *callerdat, char *dir,
                                   char *repos, char *update_dir,
                                   List *entries));
static int admin_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static const char *const admin_usage[] =
{
    "Usage: %s %s rcs-options files...\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

/* This structure is used to pass information through start_recursion.  */
struct admin_data
{
    /* Set default branch (-b).  It is "-b" followed by the value
       given, or NULL if not specified, or merely "-b" if -b is
       specified without a value.  */
    char *branch;

    /* Set comment leader (-c).  It is "-c" followed by the value
       given, or NULL if not specified.  The comment leader is
       relevant only for old versions of RCS, but we let people set it
       anyway.  */
    char *comment;

    /* Set strict locking (-L).  */
    int set_strict;

    /* Set nonstrict locking (-U).  */
    int set_nonstrict;

    /* Delete revisions (-o).  It is "-o" followed by the value specified.  */
    char *delete_revs;

    /* Keyword substitution mode (-k), e.g. "-kb".  */
    char *kflag;

    /* Description (-t).  See sanity.sh for various moanings about
       files and stdin and such.  "" if -t specified without an
       argument.  It is "-t" followed by the argument.  */
    char *desc;

    /* Interactive (-I).  Problematic with client/server.  */
    int interactive;

    /* Quiet (-q).  Not the same as the global -q option, which is a bit
       on the confusing side, perhaps.  */
    int quiet;

    /* This is the cheesy part.  It is a vector with the options which
       we don't deal with above (e.g. "-afoo" "-abar,baz").  In the future
       this presumably will be replaced by other variables which break
       out the data in a more convenient fashion.  AV as well as each of
       the strings it points to is malloc'd.  */
    int ac;
    char **av;
    int av_alloc;
};

/* Add an argument.  OPT is the option letter, e.g. 'a'.  ARG is the
   argument to that option, or NULL if omitted (whether NULL can actually
   happen depends on whether the option was specified as optional to
   getopt).  */
static void
arg_add (dat, opt, arg)
    struct admin_data *dat;
    int opt;
    char *arg;
{
    char *newelt = xmalloc ((arg == NULL ? 0 : strlen (arg)) + 3);
```

```
    strcpy (newelt, "-");
    newelt[1] = opt;
    if (arg == NULL)
        newelt[2] = '\0';
    else
        strcpy (newelt + 2, arg);

    if (dat->av_alloc == 0)
    {
        dat->av_alloc = 1;
        dat->av = (char **) xmalloc (dat->av_alloc * sizeof (*dat->av));
    }
    else if (dat->ac >= dat->av_alloc)
    {
        dat->av_alloc *= 2;
        dat->av = (char **) xrealloc (dat->av,
                                      dat->av_alloc * sizeof (*dat->av));
    }
    dat->av[dat->ac++] = newelt;
}

int
admin (argc, argv)
    int argc;
    char **argv;
{
    int err;
#ifdef CVS_ADMIN_GROUP
    struct group *grp;
    struct group *getgrnam();
#endif
    struct admin_data admin_data;
    int c;
    int i;

    if (argc <= 1)
        usage (admin_usage);

#ifdef CVS_ADMIN_GROUP
    grp = getgrnam(CVS_ADMIN_GROUP);
     /* skip usage right check if group CVS_ADMIN_GROUP does not exist */
    if (grp != NULL)
    {
        char *me = getcaller();
        char **grnam = grp->gr_mem;
        int denied = 1;

        while (*grnam)
        {
            if (strcmp(*grnam, me) == 0)
            {
                denied = 0;
                break;
            }
            grnam++;
        }

        if (denied)
            error (1, 0, "usage is restricted to members of the group %s",
                   CVS_ADMIN_GROUP);
    }
#endif

    wrap_setup ();

    memset (&admin_data, 0, sizeof admin_data);

    /* TODO: get rid of '-' switch notation in admin_data.  For
       example, admin_data->branch should be not '-bfoo' but simply 'foo'. */

    optind = 0;
    while ((c = getopt (argc, argv,
                        "+ib::c:a:A:e:l::u::LUn:N:m:o:s:t::IqxV:k:")) != -1)
    {
        switch (c)
        {
            case 'i':
                /* This has always been documented as useless in cvs.texinfo
                   and it really is--admin_fileproc silently does nothing
                   if vers->vn_user is NULL. */
                error (0, 0, "the -i option to admin is not supported");
                error (0, 0, "run add or import to create an RCS file");
                goto usage_error;

            case 'b':
                if (admin_data.branch != NULL)
                {
                    error (0, 0, "duplicate 'b' option");
                    goto usage_error;
                }
```

```
            if (optarg == NULL)
180             admin_data.branch = xstrdup ("-b");
            else
            {
                admin_data.branch = xmalloc (strlen (optarg) + 5);
                strcpy (admin_data.branch, "-b");
                strcat (admin_data.branch, optarg);
            }
            break;

        case 'c':
190         if (admin_data.comment != NULL)
            {
                error (0, 0, "duplicate 'c' option");
                goto usage_error;
            }
            admin_data.comment = xmalloc (strlen (optarg) + 5);
            strcpy (admin_data.comment, "-c");
            strcat (admin_data.comment, optarg);
            break;

200     case 'a':
            arg_add (&admin_data, 'a', optarg);
            break;

        case 'A':
            /* In the client/server case, this is cheesy because
               we just pass along the name of the RCS file, which
               then will want to exist on the server.  This is
               accidental; having the client specify a pathname on
               the server is not a design feature of the protocol.  */
210         arg_add (&admin_data, 'A', optarg);
            break;

        case 'e':
            arg_add (&admin_data, 'e', optarg);
            break;

        case 'l':
            /* Note that multiple -l options are legal.  */
            arg_add (&admin_data, 'l', optarg);
220         break;

        case 'u':
            /* Note that multiple -u options are legal.  */
            arg_add (&admin_data, 'u', optarg);
            break;

        case 'L':
            /* Probably could also complain if -L is specified multiple
               times, although RCS doesn't and I suppose it is reasonable
230            just to have it mean the same as a single -L.  */
            if (admin_data.set_nonstrict)
            {
                error (0, 0, "-U and -L are incompatible");
                goto usage_error;
            }
            admin_data.set_strict = 1;
            break;

        case 'U':
240         /* Probably could also complain if -U is specified multiple
               times, although RCS doesn't and I suppose it is reasonable
               just to have it mean the same as a single -U.  */
            if (admin_data.set_strict)
            {
                error (0, 0, "-U and -L are incompatible");
                goto usage_error;
            }
            admin_data.set_nonstrict = 1;
            break;
250
        case 'n':
            /* Mostly similar to cvs tag.  Could also be parsing
               the syntax of optarg, although for now we just pass
               it to rcs as-is.  Note that multiple -n options are
               legal.  */
            arg_add (&admin_data, 'n', optarg);
            break;

        case 'N':
260         /* Mostly similar to cvs tag.  Could also be parsing
               the syntax of optarg, although for now we just pass
               it to rcs as-is.  Note that multiple -N options are
               legal.  */
            arg_add (&admin_data, 'N', optarg);
            break;

        case 'm':
            /* Change log message.  Could also be parsing the syntax
```

```
                of optarg, although for now we just pass it to rcs
                as-is.  Note that multiple -m options are legal.  */
              arg_add (&admin_data, 'm', optarg);
              break;

            case 'o':
              /* Delete revisions.  Probably should also be parsing the
                 syntax of optarg, so that the client can give errors
                 rather than making the server take care of that.
                 Other than that I'm not sure whether it matters much
                 whether we parse it here or in admin_fileproc.

                 Note that multiple -o options are illegal, in RCS
                 as well as here.  */

              if (admin_data.delete_revs != NULL)
              {
                  error (0, 0, "duplicate '-o' option");
                  goto usage_error;
              }
              admin_data.delete_revs = xmalloc (strlen (optarg) + 5);
              strcpy (admin_data.delete_revs, "-o");
              strcat (admin_data.delete_revs, optarg);
              break;

            case 's':
              /* Note that multiple -s options are legal.  */
              arg_add (&admin_data, 's', optarg);
              break;

            case 't':
              if (admin_data.desc != NULL)
              {
                  error (0, 0, "duplicate 't' option");
                  goto usage_error;
              }
              if (optarg == NULL)
                  admin_data.desc = xstrdup ("-t");
              else
              {
                  admin_data.desc = xmalloc (strlen (optarg) + 5);
                  strcpy (admin_data.desc, "-t");
                  strcat (admin_data.desc, optarg);
              }
              break;

            case 'I':
              /* At least in RCS this can be specified several times,
                 with the same meaning as being specified once.  */
              admin_data.interactive = 1;
              break;

            case 'q':
              admin_data.quiet = 1;
              break;

            case 'x':
              error (0, 0, "the -x option has never done anything useful");
              error (0, 0, "RCS files in CVS always end in ,v");
              goto usage_error;

            case 'V':
              /* No longer supported. */
              error (0, 0, "the `-V' option is obsolete");
              break;

            case 'k':
              if (admin_data.kflag != NULL)
              {
                  error (0, 0, "duplicate '-k' option");
                  goto usage_error;
              }
              admin_data.kflag = RCS_check_kflag (optarg);
              break;
            default:
            case '?':
              /* getopt will have printed an error message.  */

            usage_error:
              /* Don't use command_name; it might be "server".  */
              error (1, 0, "specify %s -H admin for usage information",
                     program_name);
        }
    }
    argc -= optind;
    argv += optind;

    for (i = 0; i < admin_data.ac; ++i)
    {
        assert (admin_data.av[i][0] == '-');
```

```
        switch (admin_data.av[i][1])
        {
            case 'm':
            case 'l':
            case 'u':
                check_numeric (&admin_data.av[i][2], argc, argv);
                break;
            default:
                break;
        }
    }
    if (admin_data.branch != NULL)
        check_numeric (admin_data.branch + 2, argc, argv);
    if (admin_data.delete_revs != NULL)
    {
        char *p;

        check_numeric (admin_data.delete_revs + 2, argc, argv);
        p = strchr (admin_data.delete_revs + 2, ':');
        if (p != NULL && isdigit (p[1]))
            check_numeric (p + 1, argc, argv);
        else if (p != NULL && p[1] == ':' && isdigit(p[2]))
            check_numeric (p + 2, argc, argv);
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        /* We're the client side.  Fire up the remote server.  */
        start_server ();

        ign_setup ();

        /* Note that option_with_arg does not work for us, because some
           of the options must be sent without a space between the option
           and its argument.  */
        if (admin_data.interactive)
            error (1, 0, "-I option not useful with client/server");
        if (admin_data.branch != NULL)
            send_arg (admin_data.branch);
        if (admin_data.comment != NULL)
            send_arg (admin_data.comment);
        if (admin_data.set_strict)
            send_arg ("-L");
        if (admin_data.set_nonstrict)
            send_arg ("-U");
        if (admin_data.delete_revs != NULL)
            send_arg (admin_data.delete_revs);
        if (admin_data.desc != NULL)
            send_arg (admin_data.desc);
        if (admin_data.quiet)
            send_arg ("-q");
        if (admin_data.kflag != NULL)
            send_arg (admin_data.kflag);

        for (i = 0; i < admin_data.ac; ++i)
            send_arg (admin_data.av[i]);

        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, 0, 0, SEND_NO_CONTENTS);
        send_to_server ("admin\012", 0);
        err = get_responses_and_close ();
        goto return_it;
    }
#endif /* CLIENT_SUPPORT */

    lock_tree_for_write (argc, argv, 0, 0);

    err = start_recursion (admin_fileproc, (FILESDONEPROC) NULL, admin_dirproc,
                           (DIRLEAVEPROC) NULL, (void *)&admin_data,
                           argc, argv, 0,
                           W_LOCAL, 0, 0, (char *) NULL, 1);
    Lock_Cleanup ();

 return_it:
    if (admin_data.branch != NULL)
        free (admin_data.branch);
    if (admin_data.comment != NULL)
        free (admin_data.comment);
    if (admin_data.delete_revs != NULL)
        free (admin_data.delete_revs);
    if (admin_data.kflag != NULL)
        free (admin_data.kflag);
    if (admin_data.desc != NULL)
        free (admin_data.desc);
    for (i = 0; i < admin_data.ac; ++i)
        free (admin_data.av[i]);
    if (admin_data.av != NULL)
        free (admin_data.av);
```

```
    return (err);
}

/*
 * Called to run "rcs" on a particular file.
 */
/* ARGSUSED */
static int
admin_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    struct admin_data *admin_data = (struct admin_data *) callerdat;
    Vers_TS *vers;
    char *version;
    int i;
    int status = 0;
    RCSNode *rcs, *rcs2;

    vers = Version_TS (finfo, NULL, NULL, NULL, 0, 0);

    version = vers->vn_user;
    if (version == NULL)
        goto exitfunc;
    else if (strcmp (version, "0") == 0)
    {
        error (0, 0, "cannot admin newly added file `%s'", finfo->file);
        goto exitfunc;
    }

    rcs = vers->srcfile;
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    status = 0;

    if (!admin_data->quiet)
    {
        cvs_output ("RCS file: ", 0);
        cvs_output (rcs->path, 0);
        cvs_output ("\n", 1);
    }

    if (admin_data->branch != NULL)
    {
        char *branch = &admin_data->branch[2];
        if (*branch != '\0' && ! isdigit (*branch))
        {
            branch = RCS_whatbranch (rcs, admin_data->branch + 2);
            if (branch == NULL)
            {
                error (0, 0, "%s: Symbolic name %s is undefined.",
                       rcs->path, admin_data->branch + 2);
                status = 1;
            }
        }
        if (status == 0)
            RCS_setbranch (rcs, branch);
        if (branch != NULL && branch != &admin_data->branch[2])
            free (branch);
    }
    if (admin_data->comment != NULL)
    {
        if (rcs->comment != NULL)
            free (rcs->comment);
        rcs->comment = xstrdup (admin_data->comment + 2);
    }
    if (admin_data->set_strict)
        rcs->strict_locks = 1;
    if (admin_data->set_nonstrict)
        rcs->strict_locks = 0;
    if (admin_data->delete_revs != NULL)
    {
        char *s, *t, *rev1, *rev2;
        /* Set for :, clear for ::.  */
        int inclusive;
        char *t2;

        s = admin_data->delete_revs + 2;
        inclusive = 1;
        t = strchr (s, ':');
        if (t != NULL)
        {
            if (t[1] == ':')
            {
                inclusive = 0;
                t2 = t + 2;
            }
            else
                t2 = t + 1;
```

```
        }

        /* Note that we don't support '-' for ranges.  RCS considers it
           obsolete and it is problematic with tags containing '-'.  "cvs log"
           has made the same decision.  */

        if (t == NULL)
        {
            /* -orev */
            rev1 = xstrdup (s);
            rev2 = xstrdup (s);
        }
        else if (t == s)
        {
            /* -o:rev2 */
            rev1 = NULL;
            rev2 = xstrdup (t2);
        }
        else
        {
            *t = '\0';
            rev1 = xstrdup (s);
            *t = ':'; /* probably unnecessary */
            if (*t2 == '\0')
                /* -orev1: */
                rev2 = NULL;
            else
                /* -orev1:rev2 */
                rev2 = xstrdup (t2);
        }

        if (rev1 == NULL && rev2 == NULL)
        {
            /* RCS segfaults if '-o:' is given */
            error (0, 0, "no valid revisions specified in `%s' option",
                   admin_data->delete_revs);
            status = 1;
        }
        else
        {
            status |= RCS_delete_revs (rcs, rev1, rev2, inclusive);
            if (rev1)
                free (rev1);
            if (rev2)
                free (rev2);
        }
    }
    if (admin_data->desc != NULL)
    {
        free (rcs->desc);
        rcs->desc = NULL;
        if (admin_data->desc[2] == '-')
            rcs->desc = xstrdup (admin_data->desc + 3);
        else
        {
            char *descfile = admin_data->desc + 2;
            size_t bufsize = 0;
            size_t len;

            /* If -t specified with no argument, read from stdin. */
            if (*descfile == '\0')
                descfile = NULL;
            get_file (descfile, descfile, "r", &rcs->desc, &bufsize, &len);
        }
    }
    if (admin_data->kflag != NULL)
    {
        char *kflag = admin_data->kflag + 2;
        if (!rcs->expand || strcmp (rcs->expand, kflag) != 0)
        {
            if (rcs->expand)
                free (rcs->expand);
            rcs->expand = xstrdup (kflag);
        }
    }

    /* Handle miscellaneous options.  TODO: decide whether any or all
       of these should have their own fields in the admin_data
       structure. */
    for (i = 0; i < admin_data->ac; ++i)
    {
        char *arg;
        char *p, *rev, *revnum, *tag, *msg;
        char **users;
        int argc, u;
        Node *n;
        RCSVers *delta;

        arg = admin_data->av[i];
        switch (arg[1])
```

```
    {
630     case 'a': /* fall through */
        case 'e':
            line2argv (&argc, &users, arg + 2, " ,\t\n");
            if (arg[1] == 'a')
                for (u = 0; u < argc; ++u)
                    RCS_addaccess (rcs, users[u]);
            else
                for (u = 0; u < argc; ++u)
                    RCS_delaccess (rcs, users[u]);
            free_names (&argc, users);
640         break;
        case 'A':

            /* See admin-19a-admin and friends in sanity.sh for
               relative pathnames.  It makes sense to think in
               terms of a syntax which give pathnames relative to
               the repository or repository corresponding to the
               current directory or some such (and perhaps don't
               include ,v), but trying to worry about such things
               is a little pointless unless you first worry about
650            whether "cvs admin -A" as a whole makes any sense
               (currently probably not, as access lists don't
               affect the behavior of CVS).  */

            rcs2 = RCS_parsercsfile (arg + 2);
            if (rcs2 == NULL)
                error (1, 0, "cannot continue");

            p = xstrdup (RCS_getaccess (rcs2));
            line2argv (&argc, &users, p, " \t\n");
660         free (p);
            freercsnode (&rcs2);

            for (u = 0; u < argc; ++u)
                RCS_addaccess (rcs, users[u]);
            free_names (&argc, users);
            break;
        case 'n': /* fall through */
        case 'N':
            if (arg[2] == '\0')
670         {
                cvs_outerr ("missing symbolic name after ", 0);
                cvs_outerr (arg, 0);
                cvs_outerr ("\n", 1);
                break;
            }
            p = strchr (arg, ':');
            if (p == NULL)
            {
                if (RCS_deltag (rcs, arg + 2) != 0)
680             {
                    error (0, 0, "%s: Symbolic name %s is undefined.",
                           rcs->path,
                           arg + 2);
                    status = 1;
                    continue;
                }
                break;
            }
            *p = '\0';
690         tag = xstrdup (arg + 2);
            *p++ = ':';

            /* Option 'n' signals an error if this tag is already bound. */
            if (arg[1] == 'n')
            {
                n = findnode (RCS_symbols (rcs), tag);
                if (n != NULL)
                {
                    error (0, 0,
700                        "%s: symbolic name %s already bound to %s",
                           rcs->path,
                           tag, n->data);
                    status = 1;
                    free (tag);
                    continue;
                }
            }

            /* Attempt to perform the requested tagging.  */
710
            if ((*p == 0 && (rev = RCS_head (rcs)))
                || (rev = RCS_tag2rev (rcs, p))) /* tag2rev may exit */
            {
                RCS_check_tag (tag); /* exit if not a valid tag */
                RCS_settag (rcs, tag, rev);
                free (rev);
            }
            else
```

```
            {
720             error (0, 0,
                       "%s: Symbolic name or revision %s is undefined",
                       rcs->path, p);
                status = 1;
            }
            free (tag);
            break;
        case 's':
            p = strchr (arg, ':');
            if (p == NULL)
730         {
                tag = xstrdup (arg + 2);
                rev = RCS_head (rcs);
            }
            else
            {
                *p = '\0';
                tag = xstrdup (arg + 2);
                *p++ = ':';
                rev = xstrdup (p);
740         }
            revnum = RCS_gettag (rcs, rev, 0, NULL);
            free (rev);
            if (revnum != NULL)
                n = findnode (rcs->versions, revnum);
            if (revnum == NULL || n == NULL)
            {
                error (0, 0,
                       "%s: can't set state of nonexisting revision %s",
                       rcs->path,
750                    rev);
                if (revnum != NULL)
                    free (revnum);
                status = 1;
                continue;
            }
            delta = (RCSVers *) n->data;
            free (delta->state);
            delta->state = tag;
            break;
760
        case 'm':
            p = strchr (arg, ':');
            if (p == NULL)
            {
                error (0, 0, "%s: -m option lacks revision number",
                       rcs->path);
                status = 1;
                continue;
            }
770         *p = '\0';
            rev = RCS_gettag (rcs, arg + 2, 0, NULL);
            if (rev == NULL)
            {
                error (0, 0, "%s: no such revision %s", rcs->path, rev);
                status = 1;
                continue;
            }
            *p++ = ':';
            msg = p;
780
            n = findnode (rcs->versions, rev);
            delta = (RCSVers *) n->data;
            if (delta->text == NULL)
            {
                delta->text = (Deltatext *) xmalloc (sizeof (Deltatext));
                memset ((void *) delta->text, 0, sizeof (Deltatext));
            }
            delta->text->version = xstrdup (delta->version);
            delta->text->log = make_message_rcslegal (msg);
790         break;

        case 'l':
            status |= RCS_lock (rcs, arg[2] ? arg + 2 : NULL, 0);
            break;
        case 'u':
            status |= RCS_unlock (rcs, arg[2] ? arg + 2 : NULL, 0);
            break;
        default: assert(0); /* can't happen */
    }
800 }

/* TODO: reconcile the weird discrepancies between
   admin_data->quiet and quiet. */
if (status == 0)
{
    RCS_rewrite (rcs, NULL, NULL);
    if (!admin_data->quiet)
        cvs_output ("done\n", 5);
```

```
    }
    else
    {
        /* Note that this message should only occur after another
           message has given a more specific error.  The point of this
           additional message is to make it clear that the previous problems
           caused CVS to forget about the idea of modifying the RCS file.  */
        error (0, 0, "cannot modify RCS file for `%s'", finfo->file);

        /* Upon failure, we want to abandon any changes made to the
           RCS data structure.  Forcing a reparse does the trick,
           but leaks memory and is kludgey.  Should we export
           free_rcsnode_contents for this purpose? */
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);
    }

 exitfunc:
    freevers_ts (&vers);
    return status;
}

/*
 * Print a warm fuzzy message
 */
/* ARGSUSED */
static Dtype
admin_dirproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    if (!quiet)
        error (0, 0, "Administrating %s", update_dir);
    return (R_PROCESS);
}
```

A.3 buffer.c

```
/* Code for the buffer data structure.  */

#include <assert.h>
#include "cvs.h"
#include "buffer.h"

#if defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT)

/* OS/2 doesn't have EIO.  FIXME: this whole notion of turning
   a different error into EIO strikes me as pretty dubious.  */
#if !defined (EIO)
#define EIO EBADPOS
#endif

/* Linked list of available buffer_data structures.  */
static struct buffer_data *free_buffer_data;

/* Local functions.  */
static void buf_default_memory_error PROTO ((struct buffer *));
static void allocate_buffer_datas PROTO((void));
static struct buffer_data *get_buffer_data PROTO((void));

/* Initialize a buffer structure.  */

struct buffer *
buf_initialize (input, output, flush, block, shutdown, memory, closure)
     int (*input) PROTO((void *, char *, int, int, int *));
     int (*output) PROTO((void *, const char *, int, int *));
     int (*flush) PROTO((void *));
     int (*block) PROTO((void *, int));
     int (*shutdown) PROTO((void *));
     void (*memory) PROTO((struct buffer *));
     void *closure;
{
    struct buffer *buf;

    buf = (struct buffer *) xmalloc (sizeof (struct buffer));
    buf->data = NULL;
    buf->last = NULL;
    buf->nonblocking = 0;
    buf->input = input;
    buf->output = output;
    buf->flush = flush;
    buf->block = block;
    buf->shutdown = shutdown;
    buf->memory_error = memory ? memory : buf_default_memory_error;
    buf->closure = closure;
    return buf;
}

/* Free a buffer structure.  */

void
buf_free (buf)
     struct buffer *buf;
{
    if (buf->data != NULL)
    {
        buf->last->next = free_buffer_data;
        free_buffer_data = buf->data;
    }
    free (buf);
}

/* Initialize a buffer structure which is not to be used for I/O.  */

struct buffer *
buf_nonio_initialize (memory)
     void (*memory) PROTO((struct buffer *));
{
    return (buf_initialize
            ((int (*) PROTO((void *, char *, int, int, int *))) NULL,
             (int (*) PROTO((void *, const char *, int, int *))) NULL,
             (int (*) PROTO((void *))) NULL,
             (int (*) PROTO((void *, int))) NULL,
             (int (*) PROTO((void *))) NULL,
             memory,
             (void *) NULL));
}

/* Default memory error handler.  */

static void
buf_default_memory_error (buf)
     struct buffer *buf;
{
    error (1, 0, "out of memory");
}
```

```
/* Allocate more buffer_data structures.  */

static void
allocate_buffer_datas ()
{
  struct buffer_data *alc;
  char *space;
  int i;

  /* Allocate buffer_data structures in blocks of 16.  */
#define ALLOC_COUNT (16)

  alc = ((struct buffer_data *)
         malloc (ALLOC_COUNT * sizeof (struct buffer_data)));
  space = (char *) valloc (ALLOC_COUNT * BUFFER_DATA_SIZE);
  if (alc == NULL || space == NULL)
    return;
  for (i = 0; i < ALLOC_COUNT; i++, alc++, space += BUFFER_DATA_SIZE)
  {
    alc->next = free_buffer_data;
    free_buffer_data = alc;
    alc->text = space;
  }
}

/* Get a new buffer_data structure.  */

static struct buffer_data *
get_buffer_data ()
{
  struct buffer_data *ret;

  if (free_buffer_data == NULL)
  {
    allocate_buffer_datas ();
    if (free_buffer_data == NULL)
      return NULL;
  }

  ret = free_buffer_data;
  free_buffer_data = ret->next;
  return ret;
}

/* See whether a buffer is empty.  */

int
buf_empty_p (buf)
  struct buffer *buf;
{
  struct buffer_data *data;

  for (data = buf->data; data != NULL; data = data->next)
    if (data->size > 0)
      return 0;
  return 1;
}

#ifdef SERVER_FLOWCONTROL
/*
 * Count how much data is stored in the buffer. .
 * Note that each buffer is a malloc'ed chunk BUFFER_DATA_SIZE.
 */

int
buf_count_mem (buf)
  struct buffer *buf;
{
  struct buffer_data *data;
  int mem = 0;

  for (data = buf->data; data != NULL; data = data->next)
    mem += BUFFER_DATA_SIZE;

  return mem;
}
#endif /* SERVER_FLOWCONTROL */

/* Add data DATA of length LEN to BUF.  */

void
buf_output (buf, data, len)
  struct buffer *buf;
  const char *data;
  int len;
{
  if (buf->data != NULL
      && (((buf->last->text + BUFFER_DATA_SIZE)
           - (buf->last->bufp + buf->last->size))
```

```
            >= len))
    {
        memcpy (buf->last->bufp + buf->last->size, data, len);
        buf->last->size += len;
        return;
    }

    while (1)
    {
        struct buffer_data *newdata;

        newdata = get_buffer_data ();
        if (newdata == NULL)
        {
            (*buf->memory_error) (buf);
            return;
        }

        if (buf->data == NULL)
            buf->data = newdata;
        else
            buf->last->next = newdata;
        newdata->next = NULL;
        buf->last = newdata;

        newdata->bufp = newdata->text;

        if (len <= BUFFER_DATA_SIZE)
        {
            newdata->size = len;
            memcpy (newdata->text, data, len);
            return;
        }

        newdata->size = BUFFER_DATA_SIZE;
        memcpy (newdata->text, data, BUFFER_DATA_SIZE);

        data += BUFFER_DATA_SIZE;
        len -= BUFFER_DATA_SIZE;
    }

    /*NOTREACHED*/
}

/* Add a '\0' terminated string to BUF.  */

void
buf_output0 (buf, string)
    struct buffer *buf;
    const char *string;
{
    buf_output (buf, string, strlen (string));
}

/* Add a single character to BUF.  */

void
buf_append_char (buf, ch)
    struct buffer *buf;
    int ch;
{
    if (buf->data != NULL
        && (buf->last->text + BUFFER_DATA_SIZE
            != buf->last->bufp + buf->last->size))
    {
        *(buf->last->bufp + buf->last->size) = ch;
        ++buf->last->size;
    }
    else
    {
        char b;

        b = ch;
        buf_output (buf, &b, 1);
    }
}

/*
 * Send all the output we've been saving up.  Returns 0 for success or
 * errno code.  If the buffer has been set to be nonblocking, this
 * will just write until the write would block.
 */

int
buf_send_output (buf)
     struct buffer *buf;
{
    if (buf->output == NULL)
        abort ();
```

```
    while (buf->data != NULL)
270 {
        struct buffer_data *data;

        data = buf->data;

        if (data->size > 0)
        {
            int status, nbytes;

            status = (*buf->output) (buf->closure, data->bufp, data->size,
280                                  &nbytes);
            if (status != 0)
            {
                /* Some sort of error.  Discard the data, and return.  */

                buf->last->next = free_buffer_data;
                free_buffer_data = buf->data;
                buf->data = NULL;
                buf->last = NULL;

290             return status;
            }

            if (nbytes != data->size)
            {
                /* Not all the data was written out.  This is only
                   permitted in nonblocking mode.  Adjust the buffer,
                   and return.  */

                assert (buf->nonblocking);
300

                data->size -= nbytes;
                data->bufp += nbytes;

                return 0;
            }
        }

        buf->data = data->next;
        data->next = free_buffer_data;
310     free_buffer_data = data;
    }

    buf->last = NULL;

    return 0;
}

/*
 * Flush any data queued up in the buffer.  If BLOCK is nonzero, then
320 * if the buffer is in nonblocking mode, put it into blocking mode for
 * the duration of the flush.  This returns 0 on success, or an error
 * code.
 */

int
buf_flush (buf, block)
     struct buffer *buf;
     int block;
{
330 int nonblocking;
    int status;

    if (buf->flush == NULL)
        abort ();

    nonblocking = buf->nonblocking;
    if (nonblocking && block)
    {
        status = set_block (buf);
340     if (status != 0)
            return status;
    }

    status = buf_send_output (buf);
    if (status == 0)
        status = (*buf->flush) (buf->closure);

    if (nonblocking && block)
    {
350     int blockstat;

        blockstat = set_nonblock (buf);
        if (status == 0)
            status = blockstat;
    }

    return status;
}
```

```
360 /*
    * Set buffer BUF to nonblocking I/O.  Returns 0 for success or errno
    * code.
    */

   int
   set_nonblock (buf)
        struct buffer *buf;
   {
       int status;
370
       if (buf->nonblocking)
           return 0;
       if (buf->block == NULL)
           abort ();
       status = (*buf->block) (buf->closure, 0);
       if (status != 0)
           return status;
       buf->nonblocking = 1;
       return 0;
380 }

   /*
    * Set buffer BUF to blocking I/O.  Returns 0 for success or errno
    * code.
    */

   int
   set_block (buf)
        struct buffer *buf;
390 {
       int status;

       if (! buf->nonblocking)
           return 0;
       if (buf->block == NULL)
           abort ();
       status = (*buf->block) (buf->closure, 1);
       if (status != 0)
           return status;
400    buf->nonblocking = 0;
       return 0;
   }

   /*
    * Send a character count and some output.  Returns errno code or 0 for
    * success.
    *
    * Sending the count in binary is OK since this is only used on a pipe
    * within the same system.
410 */

   int
   buf_send_counted (buf)
        struct buffer *buf;
   {
       int size;
       struct buffer_data *data;

       size = 0;
420    for (data = buf->data; data != NULL; data = data->next)
           size += data->size;

       data = get_buffer_data ();
       if (data == NULL)
       {
           (*buf->memory_error) (buf);
           return ENOMEM;
       }

430    data->next = buf->data;
       buf->data = data;
       if (buf->last == NULL)
           buf->last = data;

       data->bufp = data->text;
       data->size = sizeof (int);

       *((int *) data->text) = size;

440    return buf_send_output (buf);
   }

   /*
    * Send a special count.  COUNT should be negative.  It will be
    * handled speciallyi by buf_copy_counted.  This function returns 0 or
    * an errno code.
    *
    * Sending the count in binary is OK since this is only used on a pipe
```

```
 * within the same system.
 */

int
buf_send_special_count (buf, count)
     struct buffer *buf;
     int count;
{
   struct buffer_data *data;

   data = get_buffer_data ();
   if (data == NULL)
   {
      (*buf->memory_error) (buf);
      return ENOMEM;
   }

   data->next = buf->data;
   buf->data = data;
   if (buf->last == NULL)
      buf->last = data;

   data->bufp = data->text;
   data->size = sizeof (int);

   *((int *) data->text) = count;

   return buf_send_output (buf);
}

/* Append a list of buffer_data structures to an buffer.  */

void
buf_append_data (buf, data, last)
     struct buffer *buf;
     struct buffer_data *data;
     struct buffer_data *last;
{
   if (data != NULL)
   {
      if (buf->data == NULL)
         buf->data = data;
      else
         buf->last->next = data;
      buf->last = last;
   }
}

/* Append the data on one buffer to another.  This removes the data
   from the source buffer.  */

void
buf_append_buffer (to, from)
     struct buffer *to;
     struct buffer *from;
{
   buf_append_data (to, from->data, from->last);
   from->data = NULL;
   from->last = NULL;
}

/*
 * Copy the contents of file F into buffer_data structures.  We can't
 * copy directly into an buffer, because we want to handle failure and
 * succeess differently.  Returns 0 on success, or -2 if out of
 * memory, or a status code on error.  Since the caller happens to
 * know the size of the file, it is passed in as SIZE.  On success,
 * this function sets *RETP and *LASTP, which may be passed to
 * buf_append_data.
 */

int
buf_read_file (f, size, retp, lastp)
   FILE *f;
   long size;
   struct buffer_data **retp;
   struct buffer_data **lastp;
{
   int status;

   *retp = NULL;
   *lastp = NULL;

   while (size > 0)
   {
      struct buffer_data *data;
      int get;

      data = get_buffer_data ();
      if (data == NULL)
```

```
        {
            status = -2;
            goto error_return;
        }

        if (*retp == NULL)
            *retp = data;
        else
            (*lastp)->next = data;
        data->next = NULL;
        *lastp = data;

        data->bufp = data->text;
        data->size = 0;

        if (size > BUFFER_DATA_SIZE)
            get = BUFFER_DATA_SIZE;
        else
            get = size;

        errno = EIO;
        if (fread (data->text, get, 1, f) != 1)
        {
            status = errno;
            goto error_return;
        }

        data->size += get;
        size -= get;
    }

    return 0;

 error_return:
    if (*retp != NULL)
    {
        (*lastp)->next = free_buffer_data;
        free_buffer_data = *retp;
    }
    return status;
}

/*
 * Copy the contents of file F into buffer_data structures.  We can't
 * copy directly into an buffer, because we want to handle failure and
 * succeess differently.  Returns 0 on success, or -2 if out of
 * memory, or a status code on error.  On success, this function sets
 * *RETP and *LASTP, which may be passed to buf_append_data.
 */

int
buf_read_file_to_eof (f, retp, lastp)
     FILE *f;
     struct buffer_data **retp;
     struct buffer_data **lastp;
{
    int status;

    *retp = NULL;
    *lastp = NULL;

    while (!feof (f))
    {
        struct buffer_data *data;
        int get, nread;

        data = get_buffer_data ();
        if (data == NULL)
        {
            status = -2;
            goto error_return;
        }

        if (*retp == NULL)
            *retp = data;
        else
            (*lastp)->next = data;
        data->next = NULL;
        *lastp = data;

        data->bufp = data->text;
        data->size = 0;

        get = BUFFER_DATA_SIZE;

        errno = EIO;
        nread = fread (data->text, 1, get, f);
        if (nread == 0 && !feof (f))
        {
            status = errno;
```

```
            goto error_return;
        }

        data->size = nread;
    }

    return 0;

 error_return:
    if (*retp != NULL)
    {
        (*lastp)->next = free_buffer_data;
        free_buffer_data = *retp;
    }
    return status;
}

/* Return the number of bytes in a chain of buffer_data structures.  */

int
buf_chain_length (buf)
     struct buffer_data *buf;
{
    int size = 0;
    while (buf)
    {
        size += buf->size;
        buf = buf->next;
    }
    return size;
}

/* Return the number of bytes in a buffer.  */

int
buf_length (buf)
    struct buffer *buf;
{
    return buf_chain_length (buf->data);
}

/*
 * Read an arbitrary amount of data into an input buffer.  The buffer
 * will be in nonblocking mode, and we just grab what we can.  Return
 * 0 on success, or -1 on end of file, or -2 if out of memory, or an
 * error code.  If COUNTP is not NULL, *COUNTP is set to the number of
 * bytes read.
 */

int
buf_input_data (buf, countp)
     struct buffer *buf;
     int *countp;
{
    if (buf->input == NULL)
        abort ();

    if (countp != NULL)
        *countp = 0;

    while (1)
    {
        int get;
        int status, nbytes;

        if (buf->data == NULL
            || (buf->last->bufp + buf->last->size
                == buf->last->text + BUFFER_DATA_SIZE))
        {
            struct buffer_data *data;

            data = get_buffer_data ();
            if (data == NULL)
            {
                (*buf->memory_error) (buf);
                return -2;
            }

            if (buf->data == NULL)
                buf->data = data;
            else
                buf->last->next = data;
            data->next = NULL;
            buf->last = data;

            data->bufp = data->text;
            data->size = 0;
        }

        get = ((buf->last->text + BUFFER_DATA_SIZE)
```

```
            - (buf->last->bufp + buf->last->size));

        status = (*buf->input) (buf->closure,
                                buf->last->bufp + buf->last->size,
                                0, get, &nbytes);
        if (status != 0)
            return status;

        buf->last->size += nbytes;
        if (countp != NULL)
            *countp += nbytes;

        if (nbytes < get)
        {
            /* If we did not fill the buffer, then presumably we read
               all the available data.  */
            return 0;
        }
    }

    /*NOTREACHED*/
}

/*
 * Read a line (characters up to a \012) from an input buffer.  (We
 * use \012 rather than \n for the benefit of non Unix clients for
 * which \n means something else).  This returns 0 on success, or -1
 * on end of file, or -2 if out of memory, or an error code.  If it
 * succeeds, it sets *LINE to an allocated buffer holding the contents
 * of the line.  The trailing \012 is not included in the buffer.  If
 * LENP is not NULL, then *LENP is set to the number of bytes read;
 * strlen may not work, because there may be embedded null bytes.
 */

int
buf_read_line (buf, line, lenp)
     struct buffer *buf;
     char **line;
     int *lenp;
{
    if (buf->input == NULL)
        abort ();

    *line = NULL;

    while (1)
    {
        int len, finallen = 0;
        struct buffer_data *data;
        char *nl;

        /* See if there is a newline in BUF.  */
        len = 0;
        for (data = buf->data; data != NULL; data = data->next)
        {
            nl = memchr (data->bufp, '\012', data->size);
            if (nl != NULL)
            {
                finallen = nl - data->bufp;
                len += finallen;
                break;
            }
            len += data->size;
        }

        /* If we found a newline, copy the line into a memory buffer,
           and remove it from BUF.  */
        if (data != NULL)
        {
            char *p;
            struct buffer_data *nldata;

            p = malloc (len + 1);
            if (p == NULL)
                return -2;
            *line = p;

            nldata = data;
            data = buf->data;
            while (data != nldata)
            {
                struct buffer_data *next;

                memcpy (p, data->bufp, data->size);
                p += data->size;
                next = data->next;
                data->next = free_buffer_data;
                free_buffer_data = data;
                data = next;
            }
```

```
            memcpy (p, data->bufp, finallen);
            p[finallen] = '\0';

            data->size -= finallen + 1;
            data->bufp = nl + 1;
            buf->data = data;

            if (lenp != NULL)
              *lenp = len;

            return 0;
          }

          /* Read more data until we get a newline.  */
          while (1)
          {
            int size, status, nbytes;
            char *mem;

            if (buf->data == NULL
                || (buf->last->bufp + buf->last->size
                    == buf->last->text + BUFFER_DATA_SIZE))
            {
              data = get_buffer_data ();
              if (data == NULL)
              {
                (*buf->memory_error) (buf);
                return -2;
              }

              if (buf->data == NULL)
                buf->data = data;
              else
                buf->last->next = data;
              data->next = NULL;
              buf->last = data;

              data->bufp = data->text;
              data->size = 0;
            }

            mem = buf->last->bufp + buf->last->size;
            size = (buf->last->text + BUFFER_DATA_SIZE) - mem;

            /* We need to read at least 1 byte.  We can handle up to
               SIZE bytes.  This will only be efficient if the
               underlying communication stream does its own buffering,
               or is clever about getting more than 1 byte at a time.  */
            status = (*buf->input) (buf->closure, mem, 1, size, &nbytes);
            if (status != 0)
              return status;

            buf->last->size += nbytes;

            /* Optimize slightly to avoid an unnecessary call to
               memchr.  */
            if (nbytes == 1)
            {
              if (*mem == '\012')
                break;
            }
            else
            {
              if (memchr (mem, '\012', nbytes) != NULL)
                break;
            }
          }
        }
      }

      /*
       * Extract data from the input buffer BUF.  This will read up to WANT
       * bytes from the buffer.  It will set *RETDATA to point at the bytes,
       * and set *GOT to the number of bytes to be found there.  Any buffer
       * call which uses BUF may change the contents of the buffer at *DATA,
       * so the data should be fully processed before any further calls are
       * made.  This returns 0 on success, or -1 on end of file, or -2 if
       * out of memory, or an error code.
       */

      int
      buf_read_data (buf, want, retdata, got)
          struct buffer *buf;
          int want;
          char **retdata;
          int *got;
      {
        if (buf->input == NULL)
          abort ();
```

```
900     while (buf->data != NULL && buf->data->size == 0)
        {
            struct buffer_data *next;

            next = buf->data->next;
            buf->data->next = free_buffer_data;
            free_buffer_data = buf->data;
            buf->data = next;
            if (next == NULL)
                buf->last = NULL;
910     }

        if (buf->data == NULL)
        {
            struct buffer_data *data;
            int get, status, nbytes;

            data = get_buffer_data ();
            if (data == NULL)
            {
920             (*buf->memory_error) (buf);
                return -2;
            }

            buf->data = data;
            buf->last = data;
            data->next = NULL;
            data->bufp = data->text;
            data->size = 0;

930         if (want < BUFFER_DATA_SIZE)
                get = want;
            else
                get = BUFFER_DATA_SIZE;
            status = (*buf->input) (buf->closure, data->bufp, get,
                                    BUFFER_DATA_SIZE, &nbytes);
            if (status != 0)
                return status;

            data->size = nbytes;
940     }

        *retdata = buf->data->bufp;
        if (want < buf->data->size)
        {
            *got = want;
            buf->data->size -= want;
            buf->data->bufp += want;
        }
        else
950     {
            *got = buf->data->size;
            buf->data->size = 0;
        }

        return 0;
    }

    /*
     * Copy lines from an input buffer to an output buffer.  This copies
960  * all complete lines (characters up to a newline) from INBUF to
     * OUTBUF.  Each line in OUTBUF is preceded by the character COMMAND
     * and a space.
     */

    void
    buf_copy_lines (outbuf, inbuf, command)
         struct buffer *outbuf;
         struct buffer *inbuf;
         int command;
970 {
        while (1)
        {
            struct buffer_data *data;
            struct buffer_data *nldata;
            char *nl;
            int len;

            /* See if there is a newline in INBUF.  */
            nldata = NULL;
980         nl = NULL;
            for (data = inbuf->data; data != NULL; data = data->next)
            {
                nl = memchr (data->bufp, '\n', data->size);
                if (nl != NULL)
                {
                    nldata = data;
                    break;
                }
```

```
        }

        if (nldata == NULL)
        {
            /* There are no more lines in INBUF.  */
            return;
        }

        /* Put in the command.  */
        buf_append_char (outbuf, command);
        buf_append_char (outbuf, ' ');

        if (inbuf->data != nldata)
        {
            /*
             * Simply move over all the buffers up to the one containing
             * the newline.
             */
            for (data = inbuf->data; data->next != nldata; data = data->next)
                ;
            data->next = NULL;
            buf_append_data (outbuf, inbuf->data, data);
            inbuf->data = nldata;
        }

        /*
         * If the newline is at the very end of the buffer, just move
         * the buffer onto OUTBUF.  Otherwise we must copy the data.
         */
        len = nl + 1 - nldata->bufp;
        if (len == nldata->size)
        {
            inbuf->data = nldata->next;
            if (inbuf->data == NULL)
                inbuf->last = NULL;

            nldata->next = NULL;
            buf_append_data (outbuf, nldata, nldata);
        }
        else
        {
            buf_output (outbuf, nldata->bufp, len);
            nldata->bufp += len;
            nldata->size -= len;
        }
    }
}

/*
 * Copy counted data from one buffer to another.  The count is an
 * integer, host size, host byte order (it is only used across a
 * pipe).  If there is enough data, it should be moved over.  If there
 * is not enough data, it should remain on the original buffer.  A
 * negative count is a special case.  if one is seen, *SPECIAL is set
 * to the (negative) count value and no additional data is gathered
 * from the buffer; normally *SPECIAL is set to 0.  This function
 * returns the number of bytes it needs to see in order to actually
 * copy something over.
 */

int
buf_copy_counted (outbuf, inbuf, special)
     struct buffer *outbuf;
     struct buffer *inbuf;
     int *special;
{
    *special = 0;

    while (1)
    {
        struct buffer_data *data;
        int need;
        union
        {
            char intbuf[sizeof (int)];
            int i;
        } u;
        char *intp;
        int count;
        struct buffer_data *start;
        int startoff;
        struct buffer_data *stop;
        int stopwant;

        /* See if we have enough bytes to figure out the count.  */
        need = sizeof (int);
        intp = u.intbuf;
        for (data = inbuf->data; data != NULL; data = data->next)
        {
            if (data->size >= need)
```

```
        {
1080        memcpy (intp, data->bufp, need);
            break;
        }
        memcpy (intp, data->bufp, data->size);
        intp += data->size;
        need -= data->size;
    }
    if (data == NULL)
    {
        /* We don't have enough bytes to form an integer.  */
1090    return need;
    }

    count = u.i;
    start = data;
    startoff = need;

    if (count < 0)
    {
        /* A negative COUNT is a special case meaning that we
1100       don't need any further information.  */
        stop = start;
        stopwant = 0;
    }
    else
    {
        /*
         * We have an integer in COUNT.  We have gotten all the
         * data from INBUF in all buffers before START, and we
         * have gotten STARTOFF bytes from START.  See if we have
1110     * enough bytes remaining in INBUF.
         */
        need = count - (start->size - startoff);
        if (need <= 0)
        {
            stop = start;
            stopwant = count;
        }
        else
        {
1120        for (data = start->next; data != NULL; data = data->next)
            {
                if (need <= data->size)
                    break;
                need -= data->size;
            }
            if (data == NULL)
            {
                /* We don't have enough bytes.  */
                return need;
1130        }
            stop = data;
            stopwant = need;
        }
    }

    /*
     * We have enough bytes.  Free any buffers in INBUF before
     * START, and remove STARTOFF bytes from START, so that we can
     * forget about STARTOFF.
1140 */
    start->bufp += startoff;
    start->size -= startoff;

    if (start->size == 0)
        start = start->next;

    if (stop->size == stopwant)
    {
        stop = stop->next;
1150    stopwant = 0;
    }

    while (inbuf->data != start)
    {
        data = inbuf->data;
        inbuf->data = data->next;
        data->next = free_buffer_data;
        free_buffer_data = data;
    }
1160
    /* If COUNT is negative, set *SPECIAL and get out now.  */
    if (count < 0)
    {
        *special = count;
        return 0;
    }

    /*
```

```
         * We want to copy over the bytes from START through STOP.  We
         * only want STOPWANT bytes from STOP.
         */

        if (start != stop)
        {
            /* Attach the buffers from START through STOP to OUTBUF.  */
            for (data = start; data->next != stop; data = data->next)
                ;
            inbuf->data = stop;
            data->next = NULL;
            buf_append_data (outbuf, start, data);
        }

        if (stopwant > 0)
        {
            buf_output (outbuf, stop->bufp, stopwant);
            stop->bufp += stopwant;
            stop->size -= stopwant;
        }
    }

    /*NOTREACHED*/
}

/* Shut down a buffer.  This returns 0 on success, or an errno code.  */

int
buf_shutdown (buf)
     struct buffer *buf;
{
    if (buf->shutdown)
        return (*buf->shutdown) (buf->closure);
    return 0;
}

/* The simplest type of buffer is one built on top of a stdio FILE.
   For simplicity, and because it is all that is required, we do not
   implement setting this type of buffer into nonblocking mode.  The
   closure field is just a FILE *.  */

static int stdio_buffer_input PROTO((void *, char *, int, int, int *));
static int stdio_buffer_output PROTO((void *, const char *, int, int *));
static int stdio_buffer_flush PROTO((void *));

/* Initialize a buffer built on a stdio FILE.  */

struct buffer *
stdio_buffer_initialize (fp, input, memory)
     FILE *fp;
     int input;
     void (*memory) PROTO((struct buffer *));
{
    return buf_initialize (input ? stdio_buffer_input : NULL,
                           input ? NULL : stdio_buffer_output,
                           input ? NULL : stdio_buffer_flush,
                           (int (*) PROTO((void *, int))) NULL,
                           (int (*) PROTO((void *))) NULL,
                           memory,
                           (void *) fp);
}

/* The buffer input function for a buffer built on a stdio FILE.  */

static int
stdio_buffer_input (closure, data, need, size, got)
     void *closure;
     char *data;
     int need;
     int size;
     int *got;
{
    FILE *fp = (FILE *) closure;
    int nbytes;

    /* Since stdio does its own buffering, we don't worry about
       getting more bytes than we need.  */

    if (need == 0 || need == 1)
    {
        int ch;

        ch = getc (fp);

        if (ch == EOF)
        {
            if (feof (fp))
                return -1;
            else if (errno == 0)
                return EIO;
```

```
            else
                return errno;
        }

        *data = ch;
        *got = 1;
        return 0;
    }

    nbytes = fread (data, 1, need, fp);

    if (nbytes == 0)
    {
        *got = 0;
        if (feof (fp))
            return -1;
        else if (errno == 0)
            return EIO;
        else
            return errno;
    }

    *got = nbytes;

    return 0;
}

/* The buffer output function for a buffer built on a stdio FILE.  */

static int
stdio_buffer_output (closure, data, have, wrote)
     void *closure;
     const char *data;
     int have;
     int *wrote;
{
    FILE *fp = (FILE *) closure;

    *wrote = 0;

    while (have > 0)
    {
        int nbytes;

        nbytes = fwrite (data, 1, have, fp);

        if (nbytes != have)
        {
            if (errno == 0)
                return EIO;
            else
                return errno;
        }

        *wrote += nbytes;
        have -= nbytes;
        data += nbytes;
    }

    return 0;
}

/* The buffer flush function for a buffer built on a stdio FILE.  */

static int
stdio_buffer_flush (closure)
     void *closure;
{
    FILE *fp = (FILE *) closure;

    if (fflush (fp) != 0)
    {
        if (errno == 0)
            return EIO;
        else
            return errno;
    }

    return 0;
}

/* Certain types of communication input and output data in packets,
   where each packet is translated in some fashion.  The packetizing
   buffer type supports that, given a buffer which handles lower level
   I/O and a routine to translate the data in a packet.

   This code uses two bytes for the size of a packet, so packets are
   restricted to 65536 bytes in total.

   The translation functions should just translate; they may not
```

```
   significantly increase or decrease the amount of data.  The actual
   size of the initial data is part of the translated data.  The
   output translation routine may add up to PACKET_SLOP additional
   bytes, and the input translation routine should shrink the data
   correspondingly.  */

#define PACKET_SLOP (100)

/* This structure is the closure field of a packetizing buffer.  */

struct packetizing_buffer
{
  /* The underlying buffer.  */
  struct buffer *buf;
  /* The input translation function.  Exactly one of inpfn and outfn
     will be NULL.  The input translation function should
     untranslate the data in INPUT, storing the result in OUTPUT.
     SIZE is the amount of data in INPUT, and is also the size of
     OUTPUT.  This should return 0 on success, or an errno code.  */
  int (*inpfn) PROTO((void *fnclosure, const char *input, char *output,
                      int size));
  /* The output translation function.  This should translate the
     data in INPUT, storing the result in OUTPUT.  The first two
     bytes in INPUT will be the size of the data, and so will SIZE.
     This should set *TRANSLATED to the amount of translated data in
     OUTPUT.  OUTPUT is large enough to hold SIZE + PACKET_SLOP
     bytes.  This should return 0 on success, or an errno code.  */
  int (*outfn) PROTO((void *fnclosure, const char *input, char *output,
                      int size, int *translated));
  /* A closure for the translation function.  */
  void *fnclosure;
  /* For an input buffer, we may have to buffer up data here.  */
  /* This is non-zero if the buffered data has been translated.
     Otherwise, the buffered data has not been translated, and starts
     with the two byte packet size.  */
  int translated;
  /* The amount of buffered data.  */
  int holdsize;
  /* The buffer allocated to hold the data.  */
  char *holdbuf;
  /* The size of holdbuf.  */
  int holdbufsize;
  /* If translated is set, we need another data pointer to track
     where we are in holdbuf.  If translated is clear, then this
     pointer is not used.  */
  char *holddata;
};

static int packetizing_buffer_input PROTO((void *, char *, int, int, int *));
static int packetizing_buffer_output PROTO((void *, const char *, int, int *));
static int packetizing_buffer_flush PROTO((void *));
static int packetizing_buffer_block PROTO((void *, int));
static int packetizing_buffer_shutdown PROTO((void *));

/* Create a packetizing buffer.  */

struct buffer *
packetizing_buffer_initialize (buf, inpfn, outfn, fnclosure, memory)
     struct buffer *buf;
     int (*inpfn) PROTO ((void *, const char *, char *, int));
     int (*outfn) PROTO ((void *, const char *, char *, int, int *));
     void *fnclosure;
     void (*memory) PROTO((struct buffer *));
{
  struct packetizing_buffer *pb;

  pb = (struct packetizing_buffer *) xmalloc (sizeof *pb);
  memset (pb, 0, sizeof *pb);

  pb->buf = buf;
  pb->inpfn = inpfn;
  pb->outfn = outfn;
  pb->fnclosure = fnclosure;

  if (inpfn != NULL)
  {
    /* Add PACKET_SLOP to handle larger translated packets, and
       add 2 for the count.  This buffer is increased if
       necessary.  */
    pb->holdbufsize = BUFFER_DATA_SIZE + PACKET_SLOP + 2;
    pb->holdbuf = xmalloc (pb->holdbufsize);
  }

  return buf_initialize (inpfn != NULL ? packetizing_buffer_input : NULL,
                         inpfn != NULL ? NULL : packetizing_buffer_output,
                         inpfn != NULL ? NULL : packetizing_buffer_flush,
                         packetizing_buffer_block,
                         packetizing_buffer_shutdown,
                         memory,
                         pb);
```

```
}

/* Input data from a packetizing buffer.  */

static int
packetizing_buffer_input (closure, data, need, size, got)
     void *closure;
     char *data;
     int need;
     int size;
     int *got;
{
  struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;

  *got = 0;

  if (pb->holdsize > 0 && pb->translated)
  {
    int copy;

    copy = pb->holdsize;

    if (copy > size)
    {
      memcpy (data, pb->holddata, size);
      pb->holdsize -= size;
      pb->holddata += size;
      *got = size;
      return 0;
    }

    memcpy (data, pb->holddata, copy);
    pb->holdsize = 0;
    pb->translated = 0;

    data += copy;
    need -= copy;
    size -= copy;
    *got = copy;
  }

  while (need > 0 || *got == 0)
  {
    int get, status, nread, count, tcount;
    char *bytes;
    char stackoutbuf[BUFFER_DATA_SIZE + PACKET_SLOP];
    char *inbuf, *outbuf;

    /* If we don't already have the two byte count, get it.  */
    if (pb->holdsize < 2)
    {
      get = 2 - pb->holdsize;
      status = buf_read_data (pb->buf, get, &bytes, &nread);
      if (status != 0)
      {
        /* buf_read_data can return -2, but a buffer input
           function is only supposed to return -1, 0, or an
           error code.  */
        if (status == -2)
          status = ENOMEM;
        return status;
      }

      if (nread == 0)
      {
        /* The buffer is in nonblocking mode, and we didn't
           manage to read anything.  */
        return 0;
      }

      if (get == 1)
        pb->holdbuf[1] = bytes[0];
      else
      {
        pb->holdbuf[0] = bytes[0];
        if (nread < 2)
        {
          /* We only got one byte, but we needed two.  Stash
             the byte we got, and try again.  */
          pb->holdsize = 1;
          continue;
        }
        pb->holdbuf[1] = bytes[1];
      }
      pb->holdsize = 2;
    }

    /* Read the packet.  */

    count = (((pb->holdbuf[0] & 0xff) << 8)
```

```
            + (pb->holdbuf[1] & 0xff));

  if (count + 2 > pb->holdbufsize)
  {
    char *n;

    /* We didn't allocate enough space in the initialize
       function.  */

    n = realloc (pb->holdbuf, count + 2);
    if (n == NULL)
    {
      (*pb->buf->memory_error) (pb->buf);
      return ENOMEM;
    }
    pb->holdbuf = n;
    pb->holdbufsize = count + 2;
  }

  get = count - (pb->holdsize - 2);

  status = buf_read_data (pb->buf, get, &bytes, &nread);
  if (status != 0)
  {
    /* buf_read_data can return -2, but a buffer input
       function is only supposed to return -1, 0, or an error
       code.  */
    if (status == -2)
      status = ENOMEM;
    return status;
  }

  if (nread == 0)
  {
    /* We did not get any data.  Presumably the buffer is in
       nonblocking mode.  */
    return 0;
  }

  if (nread < get)
  {
    /* We did not get all the data we need to fill the packet.
       buf_read_data does not promise to return all the bytes
       requested, so we must try again.  */
    memcpy (pb->holdbuf + pb->holdsize, bytes, nread);
    pb->holdsize += nread;
    continue;
  }

  /* We have a complete untranslated packet of COUNT bytes.  */

  if (pb->holdsize == 2)
  {
    /* We just read the entire packet (the 2 bytes in
       PB->HOLDBUF are the size).  Save a memcpy by
       translating directly from BYTES.  */
    inbuf = bytes;
  }
  else
  {
    /* We already had a partial packet in PB->HOLDBUF.  We
       need to copy the new data over to make the input
       contiguous.  */
    memcpy (pb->holdbuf + pb->holdsize, bytes, nread);
    inbuf = pb->holdbuf + 2;
  }

  if (count <= sizeof stackoutbuf)
    outbuf = stackoutbuf;
  else
  {
    outbuf = malloc (count);
    if (outbuf == NULL)
    {
      (*pb->buf->memory_error) (pb->buf);
      return ENOMEM;
    }
  }

  status = (*pb->inpfn) (pb->fnclosure, inbuf, outbuf, count);
  if (status != 0)
    return status;

  /* The first two bytes in the translated buffer are the real
     length of the translated data.  */
  tcount = ((outbuf[0] & 0xff) << 8) + (outbuf[1] & 0xff);

  if (tcount > count)
    error (1, 0, "Input translation failure");
```

```
      if (tcount > size)
1620  {
        /* We have more data than the caller has provided space
           for.  We need to save some of it for the next call.  */

        memcpy (data, outbuf + 2, size);
        *got += size;

        pb->holdsize = tcount - size;
        memcpy (pb->holdbuf, outbuf + 2 + size, tcount - size);
        pb->holddata = pb->holdbuf;
1630    pb->translated = 1;

        if (outbuf != stackoutbuf)
          free (outbuf);

        return 0;
      }

      memcpy (data, outbuf + 2, tcount);

1640  if (outbuf != stackoutbuf)
        free (outbuf);

      pb->holdsize = 0;

      data += tcount;
      need -= tcount;
      size -= tcount;
      *got += tcount;
    }
1650
    return 0;
}

/* Output data to a packetizing buffer.  */

static int
packetizing_buffer_output (closure, data, have, wrote)
     void *closure;
     const char *data;
1660 int have;
     int *wrote;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;
    char inbuf[BUFFER_DATA_SIZE + 2];
    char stack_outbuf[BUFFER_DATA_SIZE + PACKET_SLOP + 4];
    struct buffer_data *outdata;
    char *outbuf;
    int size, status, translated;

1670 if (have > BUFFER_DATA_SIZE)
    {
      /* It would be easy to malloc a buffer, but I don't think this
         case can ever arise.  */
      abort ();
    }

    inbuf[0] = (have >> 8) & 0xff;
    inbuf[1] = have & 0xff;
    memcpy (inbuf + 2, data, have);
1680
    size = have + 2;

    /* The output function is permitted to add up to PACKET_SLOP
       bytes, and we need 2 bytes for the size of the translated data.
       If we can guarantee that the result will fit in a buffer_data,
       we translate directly into one to avoid a memcpy in buf_output.  */
    if (size + PACKET_SLOP + 2 > BUFFER_DATA_SIZE)
      outbuf = stack_outbuf;
    else
1690  {
      outdata = get_buffer_data ();
      if (outdata == NULL)
      {
        (*pb->buf->memory_error) (pb->buf);
        return ENOMEM;
      }

      outdata->next = NULL;
      outdata->bufp = outdata->text;
1700
      outbuf = outdata->text;
    }

    status = (*pb->outfn) (pb->fnclosure, inbuf, outbuf + 2, size,
                           &translated);
    if (status != 0)
      return status;
```

```
    /* The output function is permitted to add up to PACKET_SLOP
       bytes.  */
    if (translated > size + PACKET_SLOP)
      abort ();

    outbuf[0] = (translated >> 8) & 0xff;
    outbuf[1] = translated & 0xff;

    if (outbuf == stack_outbuf)
      buf_output (pb->buf, outbuf, translated + 2);
    else
    {
      outdata->size = translated + 2;
      buf_append_data (pb->buf, outdata, outdata);
    }

    *wrote = have;

    /* We will only be here because buf_send_output was called on the
       packetizing buffer.  That means that we should now call
       buf_send_output on the underlying buffer.  */
    return buf_send_output (pb->buf);
}

/* Flush data to a packetizing buffer.  */

static int
packetizing_buffer_flush (closure)
     void *closure;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;

    /* Flush the underlying buffer.  Note that if the original call to
       buf_flush passed 1 for the BLOCK argument, then the buffer will
       already have been set into blocking mode, so we should always
       pass 0 here.  */
    return buf_flush (pb->buf, 0);
}

/* The block routine for a packetizing buffer.  */

static int
packetizing_buffer_block (closure, block)
     void *closure;
     int block;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;

    if (block)
      return set_block (pb->buf);
    else
      return set_nonblock (pb->buf);
}

/* Shut down a packetizing buffer.  */

static int
packetizing_buffer_shutdown (closure)
     void *closure;
{
    struct packetizing_buffer *pb = (struct packetizing_buffer *) closure;

    return buf_shutdown (pb->buf);
}

#endif /* defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT) */
```

A.4 buffer.h

```
/* Declarations concerning the buffer data structure.  */

#if defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT)

/*
 * We must read data from a child process and send it across the
 * network.  We do not want to block on writing to the network, so we
 * store the data from the child process in memory.  A BUFFER
 * structure holds the status of one communication, and uses a linked
 * list of buffer_data structures to hold data.
 */

struct buffer
{
    /* Data.  */
    struct buffer_data *data;

    /* Last buffer on data chain.  */
    struct buffer_data *last;

    /* Nonzero if the buffer is in nonblocking mode.  */
    int nonblocking;

    /* Functions must be provided to transfer data in and out of the
       buffer.  Either the input or output field must be set, but not
       both.  */

    /* Read data into the buffer DATA.  There is room for up to SIZE
       bytes.  In blocking mode, wait until some input, at least NEED
       bytes, is available (NEED may be 0 but that is the same as NEED
       == 1).  In non-blocking mode return immediately no matter how
       much input is available; NEED is ignored. Return 0 on success,
       or -1 on end of file, or an errno code.  Set the number of
       bytes read in *GOT.

       If there are a nonzero number of bytes available, less than NEED,
       followed by end of file, just read those bytes and return 0.  */
    int (*input) PROTO((void *closure, char *data, int need, int size,
                        int *got));

    /* Write data.  This should write up to HAVE bytes from DATA.
       This should return 0 on success, or an errno code.  It should
       set the number of bytes written in *WROTE.  */
    int (*output) PROTO((void *closure, const char *data, int have,
                         int *wrote));

    /* Flush any data which may be buffered up after previous calls to
       OUTPUT.  This should return 0 on success, or an errno code.  */
    int (*flush) PROTO((void *closure));

    /* Change the blocking mode of the underlying communication
       stream.  If BLOCK is non-zero, it should be placed into
       blocking mode.  Otherwise, it should be placed into
       non-blocking mode.  This should return 0 on success, or an
       errno code.  */
    int (*block) PROTO ((void *closure, int block));

    /* Shut down the communication stream.  This does not mean that it
       should be closed.  It merely means that no more data will be
       read or written, and that any final processing that is
       appropriate should be done at this point.  This may be NULL.
       It should return 0 on success, or an errno code.  This entry
       point exists for the compression code.  */
    int (*shutdown) PROTO((void *closure));

    /* This field is passed to the INPUT, OUTPUT, and BLOCK functions.  */
    void *closure;

    /* Function to call if we can't allocate memory.  */
    void (*memory_error) PROTO((struct buffer *));
};

/* Data is stored in lists of these structures.  */

struct buffer_data
{
    /* Next buffer in linked list.  */
    struct buffer_data *next;

    /*
     * A pointer into the data area pointed to by the text field.  This
     * is where to find data that has not yet been written out.
     */
    char *bufp;

    /* The number of data bytes found at BUFP.  */
    int size;
```

```
    /*
     * Actual buffer.  This never changes after the structure is
     * allocated.  The buffer is BUFFER_DATA_SIZE bytes.
     */
    char *text;
};

/* The size we allocate for each buffer_data structure.  */
#define BUFFER_DATA_SIZE (4096)

/* The type of a function passed as a memory error handler.  */
typedef void (*BUFMEMERRPROC) PROTO ((struct buffer *));

extern struct buffer *buf_initialize PROTO((int (*) (void *, char *, int,
                                                     int, int *),
                                            int (*) (void *, const char *,
                                                     int, int *),
                                            int (*) (void *),
                                            int (*) (void *, int),
                                            int (*) (void *),
                                            void (*) (struct buffer *),
                                            void *));
extern void buf_free PROTO((struct buffer *));
extern struct buffer *buf_nonio_initialize PROTO((void (*) (struct buffer *)));
extern struct buffer *stdio_buffer_initialize
  PROTO((FILE *, int, void (*) (struct buffer *)));
extern struct buffer *compress_buffer_initialize
  PROTO((struct buffer *, int, int, void (*) (struct buffer *)));
extern struct buffer *packetizing_buffer_initialize
  PROTO((struct buffer *, int (*) (void *, const char *, char *, int),
          int (*) (void *, const char *, char *, int, int *), void *,
          void (*) (struct buffer *)));
extern int buf_empty_p PROTO((struct buffer *));
extern void buf_output PROTO((struct buffer *, const char *, int));
extern void buf_output0 PROTO((struct buffer *, const char *));
extern void buf_append_char PROTO((struct buffer *, int));
extern int buf_send_output PROTO((struct buffer *));
extern int buf_flush PROTO((struct buffer *, int));
extern int set_nonblock PROTO((struct buffer *));
extern int set_block PROTO((struct buffer *));
extern int buf_send_counted PROTO((struct buffer *));
extern int buf_send_special_count PROTO((struct buffer *, int));
extern void buf_append_data PROTO((struct buffer *,
                                   struct buffer_data *,
                                   struct buffer_data *));
extern void buf_append_buffer PROTO((struct buffer *, struct buffer *));
extern int buf_read_file PROTO((FILE *, long, struct buffer_data **,
                                struct buffer_data **));
extern int buf_read_file_to_eof PROTO((FILE *, struct buffer_data **,
                                       struct buffer_data **));
extern int buf_input_data PROTO((struct buffer *, int *));
extern int buf_read_line PROTO((struct buffer *, char **, int *));
extern int buf_read_data PROTO((struct buffer *, int, char **, int *));
extern void buf_copy_lines PROTO((struct buffer *, struct buffer *, int));
extern int buf_copy_counted PROTO((struct buffer *, struct buffer *, int *));
extern int buf_chain_length PROTO((struct buffer_data *));
extern int buf_length PROTO((struct buffer *));
extern int buf_shutdown PROTO((struct buffer *));

#ifdef SERVER_FLOWCONTROL
extern int buf_count_mem PROTO((struct buffer *));
#endif /* SERVER_FLOWCONTROL */

#endif /* defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT) */
```

A.5 checkin.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Check In
 *
 * Does a very careful checkin of the file "user", and tries not to spoil its
 * modification time (to avoid needless recompilations). When RCS ID keywords
 * get expanded on checkout, however, the modification time is updated and
 * there is no good way to get around this.
 *
 * Returns non-zero on error.
 */

#include "cvs.h"
#include "fileattr.h"
#include "edit.h"

int
Checkin (type, finfo, rcs, rev, tag, options, message)
    int type;
    struct file_info *finfo;
    char *rcs;
    char *rev;
    char *tag;
    char *options;
    char *message;
{
    Vers_TS *vers;
    int set_time;
    char *tocvsPath = NULL;

    /* Hmm.  This message goes to stdout and the "foo,v  <-  foo"
       message from "ci" goes to stderr.  This doesn't make a whole
       lot of sense, but making everything go to stdout can only be
       gracefully achieved once RCS_checkin is librarified.  */
    cvs_output ("Checking in ", 0);
    cvs_output (finfo->fullname, 0);
    cvs_output (";\n", 0);

    tocvsPath = wrap_tocvs_process_file (finfo->file);
    if (!noexec)
    {
        if (tocvsPath)
        {
            if (unlink_file_dir (finfo->file) < 0)
                if (! existence_error (errno))
                    error (1, errno, "cannot remove %s", finfo->fullname);
            rename_file (tocvsPath, finfo->file);
        }
    }

    if (finfo->rcs == NULL)
        finfo->rcs = RCS_parse (finfo->file, finfo->repository);

    switch (RCS_checkin (finfo->rcs, NULL, message, rev, RCS_FLAGS_KEEPFILE))
    {
        case 0:             /* everything normal */

            /* The checkin succeeded.  If checking the file out again
               would not cause any changes, we are done.  Otherwise,
               we need to check out the file, which will change the
               modification time of the file.

               The only way checking out the file could cause any
               changes is if the file contains RCS keywords.  So we if
               we are not expanding RCS keywords, we are done.  */

            if (strcmp (options, "-V4") == 0) /* upgrade to V5 now */
                options[0] = '\0';

            /* FIXME: If PreservePermissions is on, RCS_cmp_file is
               going to call RCS_checkout into a temporary file
               anyhow.  In that case, it would be more efficient to
               call RCS_checkout here, compare the resulting files
               using xcmp, and rename if necessary.  I think this
               should be fixed in RCS_cmp_file.  */
            if ((! preserve_perms
                 && options != NULL
                 && (strcmp (options, "-ko") == 0
                     || strcmp (options, "-kb") == 0))
                || RCS_cmp_file (finfo->rcs, rev, options, finfo->file) == 0)
            {
                /* The existing file is correct.  We don't have to do
                   anything.  */
```

```
            set_time = 0;
        }
        else
        {
            /* The existing file is incorrect.  We need to check
               out the correct file contents.  */
            if (RCS_checkout (finfo->rcs, finfo->file, rev, (char *) NULL,
                              options, RUN_TTY, (RCSCHECKOUTPROC) NULL,
                              (void *) NULL) != 0)
                error (1, 0, "failed when checking out new copy of %s",
                       finfo->fullname);
            xchmod (finfo->file, 1);
            set_time = 1;
        }

        wrap_fromcvs_process_file (finfo->file);

        /*
         * If we want read-only files, muck the permissions here, before
         * getting the file time-stamp.
         */
        if (!cvswrite || fileattr_get (finfo->file, "_watched"))
            xchmod (finfo->file, 0);

        /* Re-register with the new data.  */
        vers = Version_TS (finfo, NULL, tag, NULL, 1, set_time);
        if (strcmp (vers->options, "-V4") == 0)
            vers->options[0] = '\0';
        Register (finfo->entries, finfo->file, vers->vn_rcs, vers->ts_user,
                  vers->options, vers->tag, vers->date, (char *) 0, CVSroot_directory, finfo->repository);
        history_write (type, NULL, vers->vn_rcs,
                       finfo->file, finfo->repository);

        if (tocvsPath)
            if (unlink_file_dir (tocvsPath) < 0)
                error (0, errno, "cannot remove %s", tocvsPath);

        break;

    case -1:                    /* fork failed */
        if (tocvsPath)
            if (unlink_file_dir (tocvsPath) < 0)
                error (0, errno, "cannot remove %s", tocvsPath);

        if (!noexec)
            error (1, errno, "could not check in %s -- fork failed",
                   finfo->fullname);
        return (1);

    default:                    /* ci failed */

        /* The checkin failed, for some unknown reason, so we
           print an error, and return an error.  We assume that
           the original file has not been touched.  */
        if (tocvsPath)
            if (unlink_file_dir (tocvsPath) < 0)
                error (0, errno, "cannot remove %s", tocvsPath);

        if (!noexec)
            error (0, 0, "could not check in %s", finfo->fullname);
        return (1);
    }

    /*
     * When checking in a specific revision, we may have locked the wrong
     * branch, so to be sure, we do an extra unlock here before
     * returning.
     */
    if (rev)
    {
        (void) RCS_unlock (finfo->rcs, NULL, 1);
        RCS_rewrite (finfo->rcs, NULL, NULL);
    }

#ifdef SERVER_SUPPORT
    if (server_active)
    {
        if (set_time)
            /* Need to update the checked out file on the client side.  */
            server_updated (finfo, vers, SERVER_UPDATED,
                            (mode_t) -1, (unsigned char *) NULL,
                            (struct buffer *) NULL);
        else
            server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
    }
    else
#endif
        mark_up_to_date (finfo->file);

    freevers_ts (&vers);
```

```
    return (0);
}
```

A.6 checkout.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Create Version
 *
 * "checkout" creates a "version" of an RCS repository.  This version is owned
 * totally by the user and is actually an independent copy, to be dealt with
 * as seen fit.  Once "checkout" has been called in a given directory, it
 * never needs to be called again.  The user can keep up-to-date by calling
 * "update" when he feels like it; this will supply him with a merge of his
 * own modifications and the changes made in the RCS original.  See "update"
 * for details.
 *
 * "checkout" can be given a list of directories or files to be updated and in
 * the case of a directory, will recursivley create any sub-directories that
 * exist in the repository.
 *
 * When the user is satisfied with his own modifications, the present version
 * can be committed by "commit"; this keeps the present version in tact,
 * usually.
 *
 * The call is cvs checkout [options] <module-name>...
 *
 * "checkout" creates a directory ./CVS, in which it keeps its administration,
 * in two files, Repository and Entries. The first contains the name of the
 * repository.  The second contains one line for each registered file,
 * consisting of the version number it derives from, its time stamp at
 * derivation time and its name.  Both files are normal files and can be
 * edited by the user, if necessary (when the repository is moved, e.g.)
 */

#include <assert.h>
#include "cvs.h"

static char *findslash PROTO((char *start, char *p));
static int checkout_proc PROTO((int *pargc, char **argv, char *where,
                        char *mwhere, char *mfile, int shorten,
                        int local_specified, char *omodule,
                        char *msg));
static int safe_location PROTO((void));

static const char *const checkout_usage[] =
{
   "Usage:\n  %s %s [-ANPRcflnps] [-r rev | -D date] [-d dir]\n",
   "    [-j rev1] [-j rev2] [-k kopt] modules. . .\n",
   "\t-A\tReset any sticky tags/date/kopts.\n",
   "\t-N\tDon't shorten module paths if -d specified.\n",
   "\t-P\tPrune empty directories.\n",
   "\t-R\tProcess directories recursively.\n",
   "\t-c\t\"cat\" the module database.\n",
   "\t-f\tForce a head revision match if tag/date not found.\n",
   "\t-l\tLocal directory only, not recursive\n",
   "\t-n\tDo not run module program (if any).\n",
   "\t-p\tCheck out files to standard output (avoids stickiness).\n",
   "\t-s\tLike -c, but include module status.\n",
   "\t-r rev\tCheck out revision or tag. (implies -P) (is sticky)\n",
   "\t-D date\tCheck out revisions as of date. (implies -P) (is sticky)\n",
   "\t-d dir\tCheck out into dir instead of module name.\n",
   "\t-k kopt\tUse RCS kopt -k option on checkout.\n",
   "\t-j rev\tMerge in changes made between current revision and rev.\n",
   "(Specify the --help global option for a list of other help options)\n",
   NULL
};

static const char *const export_usage[] =
{
   "Usage: %s %s [-NRfln] [-r rev | -D date] [-d dir] [-k kopt] module. . .\n",
   "\t-N\tDon't shorten module paths if -d specified.\n",
   "\t-f\tForce a head revision match if tag/date not found.\n",
   "\t-l\tLocal directory only, not recursive\n",
   "\t-R\tProcess directories recursively (default).\n",
   "\t-n\tDo not run module program (if any).\n",
   "\t-r rev\tExport revision or tag.\n",
   "\t-D date\tExport revisions as of date.\n",
   "\t-d dir\tExport into dir instead of module name.\n",
   "\t-k kopt\tUse RCS kopt -k option on checkout.\n",
   "(Specify the --help global option for a list of other help options)\n",
   NULL
};

static int checkout_prune_dirs;
static int force_tag_match = 1;
static int pipeout;
static int aflag;
```

```
static char *options = NULL;
static char *tag = NULL;
static int tag_validated = 0;
static char *date = NULL;
static char *join_rev1 = NULL;
static char *join_rev2 = NULL;
static int join_tags_validated = 0;
static char *preload_update_dir = NULL;
static char *history_name = NULL;

int
checkout (argc, argv)
    int argc;
    char **argv;
{
    int i;
    int c;
    DBM *db;
    int cat = 0, err = 0, status = 0;
    int run_module_prog = 1;
    int local = 0;
    int shorten = -1;
    char *where = NULL;
    char *valid_options;
    const char *const *valid_usage;
    enum mtype m_type;

    /*
     * A smaller subset of options are allowed for the export command, which
     * is essentially like checkout, except that it hard-codes certain
     * options to be default (like -kv) and takes care to remove the CVS
     * directory when it has done its duty
     */
    if (strcmp (command_name, "export") == 0)
    {
        m_type = EXPORT;
        valid_options = "+Nnk:d:flRQqr:D:";
        valid_usage = export_usage;
    }
    else
    {
        m_type = CHECKOUT;
        valid_options = "+ANnk:d:flRpQqcsr:D:j:P";
        valid_usage = checkout_usage;
    }

    if (argc == -1)
        usage (valid_usage);

    ign_setup ();
    wrap_setup ();

    optind = 0;
    while ((c = getopt (argc, argv, valid_options)) != -1)
    {
        switch (c)
        {
            case 'A':
                aflag = 1;
                break;
            case 'N':
                shorten = 0;
                break;
            case 'k':
                if (options)
                    free (options);
                options = RCS_check_kflag (optarg);
                break;
            case 'n':
                run_module_prog = 0;
                break;
            case 'Q':
            case 'q':
#ifdef SERVER_SUPPORT
                /* The CVS 1.5 client sends these options (in addition to
                   Global_option requests), so we must ignore them.  */
                if (!server_active)
#endif
                    error (1, 0,
                           "-q or -Q must be specified before \"%s\"",
                           command_name);
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'P':
                checkout_prune_dirs = 1;
```

```
            break;
        case 'p':
            pipeout = 1;
            run_module_prog = 0; /* don't run module prog when piping */
            noexec = 1;          /* so no locks will be created */
            break;
        case 'c':
            cat = 1;
            break;
        case 'd':
            where = optarg;
            if (shorten == -1)
                shorten = 1;
            break;
        case 's':
            status = 1;
            break;
        case 'f':
            force_tag_match = 0;
            break;
        case 'r':
            tag = optarg;
            checkout_prune_dirs = 1;
            break;
        case 'D':
            date = Make_Date (optarg);
            checkout_prune_dirs = 1;
            break;
        case 'j':
            if (join_rev2)
                error (1, 0, "only two -j options can be specified");
            if (join_rev1)
                join_rev2 = optarg;
            else
                join_rev1 = optarg;
            break;
        case '?':
        default:
            usage (valid_usage);
            break;
    }
}
argc -= optind;
argv += optind;

if (shorten == -1)
    shorten = 0;

if ((cat || status) && argc != 0)
    error (1, 0, "-c and -s must not get any arguments");

if (!(cat || status) && argc == 0)
    error (1, 0, "must specify at least one module or directory");

if (where && pipeout)
    error (1, 0, "-d and -p are mutually exclusive");

if (strcmp (command_name, "export") == 0)
{
    if (!tag && !date)
        error (1, 0, "must specify a tag or date");

    if (tag && isdigit (tag[0]))
        error (1, 0, "tag `%s' must be a symbolic tag", tag);
}

#ifdef SERVER_SUPPORT
    if (server_active && where != NULL)
    {
        server_pathname_check (where);
    }
#endif

    if (!safe_location()) {
        error(1, 0, "Cannot check out files into the repository itself");
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        int expand_modules;

        start_server ();

        ign_setup ();

        /* We have to expand names here because the "expand-modules"
           directive to the server has the side-effect of having the
           server send the check-in and update programs for the
           various modules/dirs requested.  If we turn this off and
```

```
            simply request the names of the modules and directories (as
            below in !expand_modules), those files (CVS/Checkin.prog
            or CVS/Update.prog) don't get created.  Grrr.  */

        expand_modules = (!cat && !status && !pipeout
                          && supported_request ("expand-modules"));

        if (expand_modules)
        {
            /* This is done here because we need to read responses
               from the server before we send the command checkout or
               export files. */

            client_expand_modules (argc, argv, local);
        }

        if (!run_module_prog)
            send_arg ("-n");
        if (local)
            send_arg ("-l");
        if (pipeout)
            send_arg ("-p");
        if (!force_tag_match)
            send_arg ("-f");
        if (aflag)
            send_arg("-A");
        if (!shorten)
            send_arg("-N");
        if (checkout_prune_dirs && strcmp (command_name, "export") != 0)
            send_arg("-P");
        client_prune_dirs = checkout_prune_dirs;
        if (cat)
            send_arg("-c");
        if (where != NULL)
            option_with_arg ("-d", where);
        if (status)
            send_arg("-s");
        if (options != NULL && options[0] != '\0')
            send_arg (options);
        option_with_arg ("-r", tag);
        if (date)
            client_senddate (date);
        if (join_rev1 != NULL)
            option_with_arg ("-j", join_rev1);
        if (join_rev2 != NULL)
            option_with_arg ("-j", join_rev2);

        if (expand_modules)
        {
            client_send_expansions (local, where, 1);
        }
        else
        {
            int i;
            for (i = 0; i < argc; ++i)
                send_arg (argv[i]);
            client_nonexpanded_setup ();
        }

        send_to_server (strcmp (command_name, "export") == 0 ?
                        "export\012" : "co\012",
                        0);

        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    if (cat || status)
    {
        cat_module (status);
        if (options)
            free (options);
        return (0);
    }
    db = open_module ();


    /* If we've specified something like "cvs co foo/bar baz/quux"
       don't try to shorten names.  There are a few cases in which we
       could shorten (e.g. "cvs co foo/bar foo/baz"), but we don't
       handle those yet.  Better to have an extra directory created
       than the thing checked out under the wrong directory name. */

    if (argc > 1)
        shorten = 0;


    /* If we will be calling history_write, work out the name to pass
       it.  */
    if (strcmp (command_name, "export") != 0 && !pipeout)
```

```
    {
        if (tag && date)
        {
            history_name = xmalloc (strlen (tag) + strlen (date) + 2);
            sprintf (history_name, "%s:%s", tag, date);
        }
        else if (tag)
            history_name = tag;
        else
            history_name = date;
    }

    for (i = 0; i < argc; i++)
        err += do_module (db, argv[i], m_type, "Updating", checkout_proc,
                          where, shorten, local, run_module_prog,
                          (char *) NULL);
    close_module (db);
    if (options)
        free (options);
    return (err);
}

static int
safe_location ()
{
    char *current;
    char hardpath[PATH_MAX+5];
    size_t hardpath_len;
    int  x;
    int retval;

#ifdef HAVE_READLINK
    /* FIXME-arbitrary limit: should be retrying this like xgetwd.
       But how does readlink let us know that the buffer was too small?
       (by returning sizeof hardpath - 1?).  */
    x = readlink(CVSroot_directory, hardpath, sizeof hardpath - 1);
#else
    x = -1;
#endif
    if (x == -1)
    {
        strcpy(hardpath, CVSroot_directory);
    }
    else
    {
        hardpath[x] = '\0';
    }
    current = xgetwd ();
    if (current == NULL)
        error (1, errno, "could not get working directory");
    hardpath_len = strlen (hardpath);
    if (strlen (current) >= hardpath_len
        && strncmp (current, hardpath, hardpath_len) == 0)
    {
        if (/* Current is a subdirectory of hardpath.  */
            current[hardpath_len] == '/'

            /* Current is hardpath itself.  */
            || current[hardpath_len] == '\0')
            retval = 0;
        else
            /* It isn't a problem.  For example, current is
               "/foo/cvsroot-bar" and hardpath is "/foo/cvsroot".  */
            retval = 1;
    }
    else
        retval = 1;
    free (current);
    return retval;
}

struct dir_to_build
{
    /* What to put in CVS/Repository.  */
    char *repository;
    /* The path to the directory.  */
    char *dirpath;

    /* If set, don't build the directory, just change to it.
       The caller will also want to set REPOSITORY to NULL.  */
    int just_chdir;

    struct dir_to_build *next;
};

static int build_dirs_and_chdir PROTO ((struct dir_to_build *list,
                                        int sticky));

static void build_one_dir PROTO ((char *, char *, int));
```

```
450 static void
    build_one_dir (repository, dirpath, sticky)
        char *repository;
        char *dirpath;
        int sticky;
    {
        FILE *fp;

        if (!isfile (CVSADM) && strcmp (command_name, "export") != 0)
        {
460         /* I suspect that this check could be omitted.  */
            if (!isdir (repository))
                error (1, 0, "there is no repository %s", repository);

            if (Create_Admin (".", dirpath, repository,
                              sticky ? (char *) NULL : tag,
                              sticky ? (char *) NULL : date,

                              /* FIXME?  This is a guess.  If it is important
                                 for nonbranch to be set correctly here I
470                              think we need to write it one way now and
                                 then rewrite it later via WriteTag, once
                                 we've had a chance to call RCS_nodeisbranch
                                 on each file.  */
                              0, 1))
                return;

            if (!noexec)
            {
                fp = open_file (CVSADM_ENTSTAT, "w+");
480             if (fclose (fp) == EOF)
                    error (1, errno, "cannot close %s", CVSADM_ENTSTAT);
    #ifdef SERVER_SUPPORT
                if (server_active)
                    server_set_entstat (dirpath, repository);
    #endif
            }
        }
    }

490 /*
     * process_module calls us back here so we do the actual checkout stuff
     */
    /* ARGSUSED */
    static int
    checkout_proc (pargc, argv, where_orig, mwhere, mfile, shorten,
                   local_specified, omodule, msg)
        int *pargc;
        char **argv;
        char *where_orig;
500     char *mwhere;
        char *mfile;
        int shorten;
        int local_specified;
        char *omodule;
        char *msg;
    {
        int err = 0;
        int which;
        char *cp;
510     char *repository;
        char *oldupdate = NULL;
        char *where;

        /*
         * OK, so we're doing the checkout! Our args are as follows:
         *  argc,argv contain either dir or dir followed by a list of files
         *  where contains where to put it (if supplied by checkout)
         *  mwhere contains the module name or -d from module file
         *  mfile says do only that part of the module
520      *  shorten = 1 says shorten as much as possible
         *  omodule is the original arg to do_module()
         */

        /* Set up the repository (maybe) for the bottom directory.
           Allocate more space than we need so we don't need to keep
           reallocating this string. */
        repository = xmalloc (strlen (CVSroot_directory)
                              + strlen (argv[0])
                              + (mfile == NULL ? 0 : strlen (mfile))
530                           + 10);
        (void) sprintf (repository, "%s/%s", CVSroot_directory, argv[0]);
        Sanitize_Repository_Name (repository);


        /* save the original value of preload_update_dir */
        if (preload_update_dir != NULL)
            oldupdate = xstrdup (preload_update_dir);
```

```
    /* Allocate space and set up the where variable.  We allocate more
       space than necessary here so that we don't have to keep
       reallocaing it later on. */

    where = xmalloc (strlen (argv[0])
                     + (mfile == NULL ? 0 : strlen (mfile))
                     + (mwhere == NULL ? 0 : strlen (mwhere))
                     + (where_orig == NULL ? 0 : strlen (where_orig))
                     + 10);

    /* Yes, this could be written in a less verbose way, but in this
       form it is quite easy to read.

       FIXME?  The following code that sets should probably be moved
       to do_module in modules.c, since there is similar code in
       patch.c and rtag.c. */

    if (shorten)
    {
        if (where_orig != NULL)
        {
            /* If the user has specified a directory with '-d' on the
               command line, use it preferentially, even over the '-d'
               flag in the modules file. */

            (void) strcpy (where, where_orig);
        }
        else if (mwhere != NULL)
        {
            /* Second preference is the value of mwhere, which is from
               the '-d' flag in the modules file. */

            (void) strcpy (where, mwhere);
        }
        else
        {
            /* Third preference is the directory specified in argv[0]
               which is this module'e directory in the repository. */

            (void) strcpy (where, argv[0]);
        }
    }
    else
    {
        /* Use the same preferences here, bug don't shorten -- that
           is, tack on where_orig if it exists. */

        *where = '\0';

        if (where_orig != NULL)
        {
            (void) strcat (where, where_orig);
            (void) strcat (where, "/");
        }

        /* If the -d flag in the modules file specified an absolute
           directory, let the user override it with the command-line
           -d option. */

        if ((mwhere != NULL) && (! isabsolute (mwhere)))
            (void) strcat (where, mwhere);
        else
            (void) strcat (where, argv[0]);
    }
    strip_trailing_slashes (where); /* necessary? */


    /* At this point, the user may have asked for a single file or
       directory from within a module.  In that case, we should modify
       where, repository, and argv as appropriate. */

    if (mfile != NULL)
    {
        /* The mfile variable can have one or more path elements.  If
           it has multiple elements, we want to tack those onto both
           repository and where.  The last element may refer to either
           a file or directory.  Here's what to do:

           it refers to a directory
             -> simply tack it on to where and repository
           it refers to a file
             -> munge argv to contain `basename mfile` */

        char *cp;
        char *path;


        /* Paranoia check. */
```

```
    if (mfile[strlen (mfile) - 1] == '/')
    {
        error (0, 0, "checkout_proc: trailing slash on mfile (%s)!",
               mfile);
    }

    /* Does mfile have multiple path elements? */

    cp = strrchr (mfile, '/');
    if (cp != NULL)
    {
        *cp = '\0';
        (void) strcat (repository, "/");
        (void) strcat (repository, mfile);
        (void) strcat (where, "/");
        (void) strcat (where, mfile);
        mfile = cp + 1;
    }

    /* Now mfile is a single path element. */

    path = xmalloc (strlen (repository) + strlen (mfile) + 5);
    (void) sprintf (path, "%s/%s", repository, mfile);
    if (isdir (path))
    {
        /* It's a directory, so tack it on to repository and
           where, as we did above. */

        (void) strcat (repository, "/");
        (void) strcat (repository, mfile);
        (void) strcat (where, "/");
        (void) strcat (where, mfile);
    }
    else
    {
        /* It's a file, which means we have to screw around with
           argv. */

        int i;

        /* Paranoia check. */

        if (*pargc > 1)
        {
            error (0, 0, "checkout_proc: trashing argv elements!");
            for (i = 1; i < *pargc; i++)
            {
                error (0, 0, "checkout_proc: argv[%d] `%s'",
                       i, argv[i]);
            }
        }

        for (i = 1; i < *pargc; i++)
            free (argv[i]);
        argv[1] = xstrdup (mfile);
        (*pargc) = 2;
    }
    free (path);
}

if (preload_update_dir != NULL)
{
    preload_update_dir =
        xrealloc (preload_update_dir,
                  strlen (preload_update_dir) + strlen (where) + 5);
    strcat (preload_update_dir, "/");
    strcat (preload_update_dir, where);
}
else
    preload_update_dir = xstrdup (where);

/*
 * At this point, where is the directory we want to build, repository is
 * the repository for the lowest level of the path.
 *
 * We need to tell build_dirs not only the path we want it to
 * build, but also the repositories we want it to populate the
 * path with.  To accomplish this, we walk the path backwards, one
 * pathname component at a time, constucting a linked list of
 * struct dir_to_build.
 */

/*
 * If we are sending everything to stdout, we can skip a whole bunch of
 * work from here
 */
if (!pipeout)
```

```
    {
720     struct dir_to_build *head;
        char *reposcopy;

        if (strncmp (repository, CVSroot_directory,
                     strlen (CVSroot_directory)) != 0)
            error (1, 0, "\
internal error: %s doesn't start with %s in checkout_proc",
                   repository, CVSroot_directory);

        /* We always create at least one directory, which corresponds to
730        the entire strings for WHERE and REPOSITORY.  */
        head = (struct dir_to_build *) xmalloc (sizeof (struct dir_to_build));
        /* Special marker to indicate that we don't want build_dirs_and_chdir
           to create the CVSADM directory for us.  */
        head->repository = NULL;
        head->dirpath = xstrdup (where);
        head->next = NULL;
        head->just_chdir = 0;


740     /* Make a copy of the repository name to play with. */
        reposcopy = xstrdup (repository);

        /* FIXME: this should be written in terms of last_component
           instead of hardcoding '/'.  This presumably affects OS/2,
           NT, &c, if the user specifies '\'.  Likewise for the call
           to findslash.  */
        cp = where + strlen (where);
        while (1)
        {
750         struct dir_to_build *new;

            cp = findslash (where, cp - 1);
            if (cp == NULL)
                break;          /* we're done */

            new = (struct dir_to_build *)
                xmalloc (sizeof (struct dir_to_build));
            new->dirpath = xmalloc (strlen (where));

760         /* If the user specified an absolute path for where, the
               last path element we create should be the top-level
               directory. */

            if (cp - where)
            {
                strncpy (new->dirpath, where, cp - where);
                new->dirpath[cp - where] = '\0';
            }
            else
770         {
                /* where should always be at least one character long. */
                assert (strlen (where));
                strcpy (new->dirpath, "/");
            }
            new->next = head;
            head = new;

            /* If where consists of multiple pathname components,
               then we want to just cd into it, without creating
780            directories or modifying CVS directories as we go.
               In CVS 1.9 and earlier, the code actually does a
               CVS_CHDIR up-front; I'm not going to try to go back
               to that exact code but this is somewhat similar
               in spirit.  */
            if (where_orig != NULL
                && cp - where < strlen (where_orig))
            {
                new->repository = NULL;
                new->just_chdir = 1;
790             continue;
            }

            new->just_chdir = 0;

            /* Now figure out what repository directory to generate.
               The most complete case would be something like this:

               The modules file contains
                 foo -d bar/baz quux
800
               The command issued was:
                 cvs co -d what/ever -N foo

               The results in the CVS/Repository files should be:
                 .     -> (don't touch CVS/Repository)
                          (I think this case might be buggy currently)
                 what  -> (don't touch CVS/Repository)
                 ever  -> .          (same as "cd what/ever; cvs co -N foo")
```

```
                bar   -> Emptydir    (generated dir - not in repos)
                baz   -> quux        (finally!) */

            if (strcmp (reposcopy, CVSroot_directory) == 0)
            {
                /* We can't walk up past CVSROOT.  Instead, the
                   repository should be Emptydir. */
                new->repository = emptydir_name ();
            }
            else
            {
                if ((where_orig != NULL)
                    && (strcmp (new->dirpath, where_orig) == 0))
                {
                    /* It's the case that the user specified a
                     * destination directory with the "-d" flag.  The
                     * repository in this directory should be "."
                     * since the user's command is equivalent to:
                     *
                     *   cd <dir>; cvs co blah   */

                    strcpy (reposcopy, CVSroot_directory);
                    goto allocate_repos;
                }
                else if (mwhere != NULL)
                {
                    /* This is a generated directory, so point to
                       CVSNULLREPOS. */

                    new->repository = emptydir_name ();
                }
                else
                {
                    /* It's a directory in the repository! */

                    char *rp = strrchr (reposcopy, '/');

                    /* We'll always be below CVSROOT, but check for
                       paranoia's sake. */
                    if (rp == NULL)
                        error (1, 0,
                               "internal error: %s doesn't contain a slash",
                               reposcopy);

                    *rp = '\0';

                allocate_repos:
                    new->repository = xmalloc (strlen (reposcopy) + 5);
                    (void) strcpy (new->repository, reposcopy);

                    if (strcmp (reposcopy, CVSroot_directory) == 0)
                    {
                        /* Special case - the repository name needs
                           to be "/path/to/repos/." (the trailing dot
                           is important).  We might be able to get rid
                           of this after the we check out the other
                           code that handles repository names. */
                        (void) strcat (new->repository, "/.");
                    }
                }
            }
        }

        /* clean up */
        free (reposcopy);

        {
            int where_is_absolute = isabsolute (where);

            /* The top-level CVSADM directory should always be
               CVSroot_directory.  Create it, but only if WHERE is
               relative.  If WHERE is absolute, our current directory
               may not have a thing to do with where the sources are
               being checked out.  If it does, build_dirs_and_chdir
               will take care of creating adm files here. */
            /* FIXME: checking where_is_absolute is a horrid kludge;
               I suspect we probably can just skip the call to
               build_one_dir whenever the -d command option was specified
               to checkout. */

            if (! where_is_absolute && top_level_admin)
            {
                /* It may be argued that we shouldn't set any sticky
                   bits for the top-level repository.  FIXME? */
                build_one_dir (CVSroot_directory, ".", *pargc <= 1);

#ifdef SERVER_SUPPORT
                /* We _always_ want to have a top-level admin
                   directory.  If we're running in client/server mode,
                   send a "Clear-static-directory" command to make
```

```
                sure it is created on the client side.  (See 5.10
                in cvsclient.dvi to convince yourself that this is
                OK.)  If this is a duplicate command being sent, it
                will be ignored on the client side.  */

              if (server_active)
                  server_clear_entstat (".", CVSroot_directory);
#endif
          }


          /* Build dirs on the path if necessary and leave us in the
             bottom directory (where if where was specified) doesn't
             contain a CVS subdir yet, but all the others contain
             CVS and Entries.Static files */

          if (build_dirs_and_chdir (head, *pargc <= 1) != 0)
          {
              error (0, 0, "ignoring module %s", omodule);
              err = 1;
              goto out;
          }
      }

      /* set up the repository (or make sure the old one matches) */
      if (!isfile (CVSADM))
      {
          FILE *fp;

          if (!noexec && *pargc > 1)
          {
              /* I'm not sure whether this check is redundant.  */
              if (!isdir (repository))
                  error (1, 0, "there is no repository %s", repository);

              Create_Admin (".", preload_update_dir, repository,
                            (char *) NULL, (char *) NULL, 0, 0);
              fp = open_file (CVSADM_ENTSTAT, "w+");
              if (fclose(fp) == EOF)
                  error(1, errno, "cannot close %s", CVSADM_ENTSTAT);
#ifdef SERVER_SUPPORT
              if (server_active)
                  server_set_entstat (where, repository);
#endif
          }
          else
          {
              /* I'm not sure whether this check is redundant.  */
              if (!isdir (repository))
                  error (1, 0, "there is no repository %s", repository);

              Create_Admin (".", preload_update_dir, repository, tag, date,

                            /* FIXME?  This is a guess.  If it is important
                               for nonbranch to be set correctly here I
                               think we need to write it one way now and
                               then rewrite it later via WriteTag, once
                               we've had a chance to call RCS_nodeisbranch
                               on each file.  */
                            0, 0);
          }
      }
      else
      {
          char *repos;

          /* get the contents of the previously existing repository */
          repos = Name_Repository ((char *) NULL, preload_update_dir);
          if (fncmp (repository, repos) != 0)
          {
              error (0, 0, "existing repository %s does not match %s",
                     repos, repository);
              error (0, 0, "ignoring module %s", omodule);
              free (repos);
              err = 1;
              goto out;
          }
          free (repos);
      }
  }

  /*
   * If we are going to be updating to stdout, we need to cd to the
   * repository directory so the recursion processor can use the current
   * directory as the place to find repository information
   */
  if (pipeout)
  {
      if ( CVS_CHDIR (repository) < 0)
      {
```

```
            error (0, errno, "cannot chdir to %s", repository);
            err = 1;
            goto out;
        }
        which = W_REPOS;
        if (tag != NULL && !tag_validated)
        {
            tag_check_valid (tag, *pargc - 1, argv + 1, 0, aflag, NULL);
            tag_validated = 1;
        }
    }
    else
    {
        which = W_LOCAL | W_REPOS;
        if (tag != NULL && !tag_validated)
        {
            tag_check_valid (tag, *pargc - 1, argv + 1, 0, aflag,
                             repository);
            tag_validated = 1;
        }
    }

    if (tag != NULL || date != NULL || join_rev1 != NULL)
        which |= W_ATTIC;

    if (! join_tags_validated)
    {
        if (join_rev1 != NULL)
            tag_check_valid_join (join_rev1, *pargc - 1, argv + 1, 0, aflag,
                                  repository);
        if (join_rev2 != NULL)
            tag_check_valid_join (join_rev2, *pargc - 1, argv + 1, 0, aflag,
                                  repository);
        join_tags_validated = 1;
    }

    /*
     * if we are going to be recursive (building dirs), go ahead and call the
     * update recursion processor.  We will be recursive unless either local
     * only was specified, or we were passed arguments
     */
    if (!(local_specified || *pargc > 1))
    {
        if (strcmp (command_name, "export") != 0 && !pipeout)
            history_write ('O', preload_update_dir, history_name, where,
                           repository);
        else if (strcmp (command_name, "export") == 0 && !pipeout)
            history_write ('E', preload_update_dir, tag ? tag : date, where,
                           repository);
        err += do_update (0, (char **) NULL, options, tag, date,
                          force_tag_match, 0 /* !local */ ,
                          1 /* update -d */ , aflag, checkout_prune_dirs,
                          pipeout, which, join_rev1, join_rev2,
                          preload_update_dir);
        goto out;
    }

    if (!pipeout)
    {
        int i;
        List *entries;

        /* we are only doing files, so register them */
        entries = Entries_Open (0, NULL);
        for (i = 1; i < *pargc; i++)
        {
            char *line;
            Vers_TS *vers;
            struct file_info finfo;

            memset (&finfo, 0, sizeof finfo);
            finfo.file = argv[i];
            /* Shouldn't be used, so set to arbitrary value.  */
            finfo.update_dir = NULL;
            finfo.fullname = argv[i];
            finfo.repository = repository;
            finfo.entries = entries;
            /* The rcs slot is needed to get the options from the RCS
               file */
            finfo.rcs = RCS_parse (finfo.file, repository);

            vers = Version_TS (&finfo, options, tag, date,
                               force_tag_match, 0);
            if (vers->ts_user == NULL)
            {
                line = xmalloc (strlen (finfo.file) + 15);
                (void) sprintf (line, "Initial %s", finfo.file);
                Register (entries, finfo.file,
                          vers->vn_rcs ? vers->vn_rcs : "0",
                          line, vers->options, vers->tag,
```

```
                        vers->date, (char *) 0, CVSroot_directory, finfo.repository);
1080                free (line);
                }
                freevers_ts (&vers);
                freercsnode (&finfo.rcs);
            }

            Entries_Close (entries);
        }

        /* Don't log "export", just regular "checkouts" */
1090    if (strcmp (command_name, "export") != 0 && !pipeout)
            history_write ('O', preload_update_dir, history_name, where,
                           repository);

        /* go ahead and call update now that everything is set */
        err += do_update (*pargc - 1, argv + 1, options, tag, date,
                          force_tag_match, local_specified, 1 /* update -d */,
                          aflag, checkout_prune_dirs, pipeout, which, join_rev1,
                          join_rev2, preload_update_dir);
    out:
1100    free (preload_update_dir);
        preload_update_dir = oldupdate;
        free (where);
        free (repository);
        return (err);
    }

    static char *
    findslash (start, p)
        char *start;
1110    char *p;
    {
        while (p >= start && *p != '/')
            p--;
        /* FIXME: indexing off the start of the array like this is *NOT*
           OK according to ANSI, and will break some of the time on certain
           segmented architectures. */
        if (p < start)
            return (NULL);
        else
1120        return (p);
    }

    /* Return a newly malloc'd string containing a pathname for CVSNULLREPOS,
       and make sure that it exists. If there is an error creating the
       directory, give a fatal error. Otherwise, the directory is guaranteed
       to exist when we return. */
    char *
    emptydir_name ()
    {
1130    char *repository;

        repository = xmalloc (strlen (CVSroot_directory)
                              + sizeof (CVSROOTADM)
                              + sizeof (CVSNULLREPOS)
                              + 10);
        (void) sprintf (repository, "%s/%s/%s", CVSroot_directory,
                        CVSROOTADM, CVSNULLREPOS);
        if (!isfile (repository))
        {
1140        mode_t omask;
            omask = umask (cvsumask);
            if (CVS_MKDIR (repository, 0777) < 0)
                error (1, errno, "cannot create %s", repository);
            (void) umask (omask);
        }
        return repository;
    }

    /* Build all the dirs along the path to DIRS with CVS subdirs with appropriate
1150   repositories. If ->repository is NULL, do not create a CVSADM directory
       for that subdirectory; just CVS_CHDIR into it. */
    static int
    build_dirs_and_chdir (dirs, sticky)
        struct dir_to_build *dirs;
        int sticky;
    {
        int retval = 0;
        struct dir_to_build *nextdir;

1160    while (dirs != NULL)
        {
            char *dir = last_component (dirs->dirpath);

            if (!dirs->just_chdir)
            {
                mkdir_if_needed (dir);
                Subdir_Register (NULL, NULL, dir);
            }
```

```
1170        if (CVS_CHDIR (dir) < 0)
            {
                error (0, errno, "cannot chdir to %s", dir);
                retval = 1;
                goto out;
            }
            if (dirs->repository != NULL)
            {
                build_one_dir (dirs->repository, dirs->dirpath, sticky);
                free (dirs->repository);
1180        }
            nextdir = dirs->next;
            free (dirs->dirpath);
            free (dirs);
            dirs = nextdir;
        }

    out:
        return retval;
    }
```

A.7 classify.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 */

#include "cvs.h"

static void sticky_ck PROTO ((struct file_info *finfo, int aflag,
                              Vers_TS * vers));

/*
 * Classify the state of a file
 */
Ctype
Classify_File (finfo, tag, date, options, force_tag_match, aflag, versp,
               pipeout)
    struct file_info *finfo;
    char *tag;
    char *date;

    /* Keyword expansion options.  Can be either NULL or "" to
       indicate none are specified here.  */
    char *options;

    int force_tag_match;
    int aflag;
    Vers_TS **versp;
    int pipeout;
{
    Vers_TS *vers;
    Ctype ret;

    /* get all kinds of good data about the file */
    vers = Version_TS (finfo, options, tag, date,
                       force_tag_match, 0);

    if (vers->vn_user == NULL)
    {
        /* No entry available, ts_rcs is invalid */
        if (vers->vn_rcs == NULL)
        {
            /* there is no RCS file either */
            if (vers->ts_user == NULL)
            {
                /* there is no user file */
                /* FIXME: Why do we skip this message if vers->tag or
                   vers->date is set?  It causes "cvs update -r tag98 foo"
                   to silently do nothing, which is seriously confusing
                   behavior.  "cvs update foo" gives this message, which
                   is what I would expect. */
                if (!force_tag_match || !(vers->tag || vers->date))
                    if (!really_quiet)
                        error (0, 0, "nothing known about %s", finfo->fullname);
                ret = T_UNKNOWN;
            }
            else
            {
                /* there is a user file */
                /* FIXME: Why do we skip this message if vers->tag or
                   vers->date is set?  It causes "cvs update -r tag98 foo"
                   to silently do nothing, which is seriously confusing
                   behavior.  "cvs update foo" gives this message, which
                   is what I would expect. */
                if (!force_tag_match || !(vers->tag || vers->date))
                    if (!really_quiet)
                        error (0, 0, "use `%s add' to create an entry for %s",
                               program_name, finfo->fullname);
                ret = T_UNKNOWN;
            }
        }
        else if (RCS_isdead (vers->srcfile, vers->vn_rcs))
        {
            if (vers->ts_user == NULL)
                ret = T_UPTODATE;
            else
            {
                error (0, 0, "use `%s add' to create an entry for %s",
                       program_name, finfo->fullname);
                ret = T_UNKNOWN;
            }
        }
        else
        {
            /* there is an rcs file */
```

```
90          if (vers->ts_user == NULL)
            {
                /* There is no user file; needs checkout */
              if (vers->vn_remote)
                ret = T_REMOTE;
              else
                ret = T_CHECKOUT;
            }
            else
            {
100             if (pipeout)
                {
                    /*
                     * The user file doesn't necessarily have anything
                     * to do with this.
                     */
                    ret = T_CHECKOUT;
                }
                /*
                 * There is a user file; print a warning and add it to the
110              * conflict list, only if it is indeed different from what we
                 * plan to extract
                 */
                else if (No_Difference (finfo, vers))
                {
                    /* the files were different so it is a conflict */
                    if (!really_quiet)
                        error (0, 0, "move away %s; it is in the way",
                               finfo->fullname);
                    ret = T_CONFLICT;
120             }
                else
                    /* since there was no difference, still needs checkout */
                    ret = T_CHECKOUT;
            }
        }
    }
    else if (strcmp (vers->vn_user, "0") == 0)
    {
        /* An entry for a new-born file; ts_rcs is dummy */
130
        if (vers->ts_user == NULL)
        {
            /*
             * There is no user file, but there should be one; remove the
             * entry
             */
            if (!really_quiet)
                error (0, 0, "warning: new-born %s has disappeared", finfo->fullname);
            ret = T_REMOVE_ENTRY;
140     }
        else
        {
            /* There is a user file */

            if (vers->vn_rcs == NULL)
                /* There is no RCS file, added file */
                ret = T_ADDED;
            else if (RCS_isdead (vers->srcfile, vers->vn_rcs))
                /* we are resurrecting. */
150             ret = T_ADDED;
            else
            {
                if (vers->srcfile->flags & INATTIC
                    && vers->srcfile->flags & VALID)
                {
                    /* This file has been added on some branch other than
                       the one we are looking at.  In the branch we are
                       looking at, the file was already valid.  */
                    if (!really_quiet)
160                     error (0, 0,
                               "\
conflict: %s has been added, but already exists",
                               finfo->fullname);
                }
                else
                {
                    /*
                     * There is an RCS file, so someone else must have checked
                     * one in behind our back; conflict
170                  */
                    if (!really_quiet)
                        error (0, 0,
                               "\
conflict: %s created independently by second party",
                               finfo->fullname);
                }
                ret = T_CONFLICT;
            }
```

```
    }
}
else if (vers->vn_user[0] == '-')
{
    /* An entry for a removed file, ts_rcs is invalid */

    if (vers->ts_user == NULL)
    {
        /* There is no user file (as it should be) */

        if (vers->vn_rcs == NULL
            || RCS_isdead (vers->srcfile, vers->vn_rcs))
        {

            /*
             * There is no RCS file; this is all-right, but it has been
             * removed independently by a second party; remove the entry
             */
            ret = T_REMOVE_ENTRY;
        }
        else if (vers->vn_rcs == NULL
                 ? vers->vn_user[1] == '\0'
                 : strcmp (vers->vn_rcs, vers->vn_user + 1) == 0)
            /*
             * The RCS file is the same version as the user file was, and
             * that's OK; remove it
             */
            ret = T_REMOVED;
        else
        {

            /*
             * The RCS file is a newer version than the removed user file
             * and this is definitely not OK; make it a conflict.
             */
            if (!really_quiet)
                error (0, 0,
                       "conflict: removed %s was modified by second party",
                       finfo->fullname);
            ret = T_CONFLICT;
        }
    }
    else
    {
        /* The user file shouldn't be there */
        if (!really_quiet)
            error (0, 0, "%s should be removed and is still there",
                   finfo->fullname);
        ret = T_REMOVED;
    }
}
else
{
    /* A normal entry, TS_Rcs is valid */
    if (vers->vn_rcs == NULL)
    {
        /* There is no RCS file */

        if (vers->ts_user == NULL)
        {
            /* There is no user file, so just remove the entry */
            if (!really_quiet)
                error (0, 0, "warning: %s is not (any longer) pertinent",
                       finfo->fullname);
            ret = T_REMOVE_ENTRY;
        }
        else if (strcmp (vers->ts_user, vers->ts_rcs) == 0)
        {

            /*
             * The user file is still unmodified, so just remove it from
             * the entry list
             */
            if (!really_quiet)
                error (0, 0, "%s is no longer in the repository",
                       finfo->fullname);
            ret = T_REMOVE_ENTRY;
        }
        else
        {
            /*
             * The user file has been modified and since it is no longer
             * in the repository, a conflict is raised
             */
            if (No_Difference (finfo, vers))
            {
                /* they are different -> conflict */
                if (!really_quiet)
                    error (0, 0,
              "conflict: %s is modified but no longer in the repository",
```

```
                        finfo->fullname);
                ret = T_CONFLICT;
            }
            else
            {
                /* they weren't really different */
                if (!really_quiet)
                    error (0, 0,
                           "warning: %s is not (any longer) pertinent",
                           finfo->fullname);
                ret = T_REMOVE_ENTRY;
            }
        }
    }
    else if (strcmp (vers->vn_rcs, vers->vn_user) == 0)
    {
        /* The RCS file is the same version as the user file */

        if (vers->ts_user == NULL)
        {

            /*
             * There is no user file, so note that it was lost and
             * extract a new version
             */
            /* Comparing the command_name against "update", in
               addition to being an ugly way to operate, means
               that this message does not get printed by the
               server.  That might be considered just a straight
               bug, although there is one subtlety: that case also
               gets hit when a patch fails and the client fetches
               a file.  I'm not sure there is currently any way
               for the server to distinguish those two cases.  */
            if (strcmp (command_name, "update") == 0)
                if (!really_quiet)
                    error (0, 0, "warning: %s was lost", finfo->fullname);
            ret = T_CHECKOUT;
        }
        else if (strcmp (vers->ts_user, vers->ts_rcs) == 0)
        {

            /*
             * The user file is still unmodified, so nothing special at
             * all to do - no lists updated, unless the sticky -k option
             * has changed.  If the sticky tag has changed, we just need
             * to re-register the entry
             */
            /* TODO: decide whether we need to check file permissions
               for a mismatch, and return T_CONFLICT if so. */
            if (vers->entdata->options &&
                strcmp (vers->entdata->options, vers->options) != 0)
                ret = T_CHECKOUT;
            else
            {
                sticky_ck (finfo, aflag, vers);
                ret = T_UPTODATE;
            }
        }
        else
        {

            /*
             * The user file appears to have been modified, but we call
             * No_Difference to verify that it really has been modified
             */
            if (No_Difference (finfo, vers))
            {

                /*
                 * they really are different; modified if we aren't
                 * changing any sticky -k options, else needs merge
                 */
#ifdef XXX_FIXME_WHEN_RCSMERGE_IS_FIXED
                if (strcmp (vers->entdata->options ?
                            vers->entdata->options : "", vers->options) == 0)
                    ret = T_MODIFIED;
                else
                    ret = T_NEEDS_MERGE;
#else
                ret = T_MODIFIED;
                sticky_ck (finfo, aflag, vers);
#endif
            }
            else
            {
                /* file has not changed; check out if -k changed */
                if (strcmp (vers->entdata->options ?
                            vers->entdata->options : "", vers->options) != 0)
                {
                    ret = T_CHECKOUT;
```

```
                }
                else
                {

                    /*
                     * else -> note that No_Difference will Register the
                     * file already for us, using the new tag/date. This
                     * is the desired behaviour
                     */
                    ret = T_UPTODATE;
                }
            }
        }
    }
    else
    {
        /* The RCS file is a newer version than the user file */

        if (vers->ts_user == NULL)
        {
            /* There is no user file, so just get it */

            /* See comment at other "update" compare, for more
               thoughts on this comparison.  */
            if (strcmp (command_name, "update") == 0)
                if (!really_quiet)
                    error (0, 0, "warning: %s was lost", finfo->fullname);
            ret = T_CHECKOUT;
        }
        else if (strcmp (vers->ts_user, vers->ts_rcs) == 0)
        {

            /*
             * The user file is still unmodified, so just get it as well
             */
#ifdef SERVER_SUPPORT
            if (vers->vn_remote != NULL) {
              ret = T_REMOTE;
            } else
              if (strcmp (vers->entdata->options ?
                          vers->entdata->options : "", vers->options) != 0
                || (vers->srcfile != NULL
                    && (vers->srcfile->flags & INATTIC) != 0))
                ret = T_CHECKOUT;
              else
                ret = T_PATCH;
#else
              ret = T_CHECKOUT;
#endif
        }
        else
        {
            if (No_Difference (finfo, vers))
                /* really modified, needs to merge */
                ret = T_NEEDS_MERGE;
#ifdef SERVER_SUPPORT
              else if (vers->vn_remote != NULL) {
                ret = T_REMOTE;
              } else if ((strcmp (vers->entdata->options ?
                                  vers->entdata->options : "", vers->options)
                          != 0)
                         || (vers->srcfile != NULL
                             && (vers->srcfile->flags & INATTIC) != 0))
                /* not really modified, check it out */
                ret = T_CHECKOUT;
              else
                ret = T_PATCH;
#else
              else
                /* not really modified, check it out */
                ret = T_CHECKOUT;
#endif
        }
    }

    /* free up the vers struct, or just return it */
    if (versp != (Vers_TS **) NULL)
        *versp = vers;
    else
        freevers_ts (&vers);

    /* return the status of the file */
    return (ret);
}

static void
sticky_ck (finfo, aflag, vers)
    struct file_info *finfo;
    int aflag;
```

```
    Vers_TS *vers;
{
    if (aflag || vers->tag || vers->date)
    {
        char *enttag = vers->entdata->tag;
        char *entdate = vers->entdata->date;

        if ((enttag && vers->tag && strcmp (enttag, vers->tag)) ||
            ((enttag && !vers->tag) || (!enttag && vers->tag)) ||
            (entdate && vers->date && strcmp (entdate, vers->date)) ||
            ((entdate && !vers->date) || (!entdate && vers->date)))
        {
            Register (finfo->entries, finfo->file, vers->vn_user, vers->ts_rcs,
                      vers->options, vers->tag, vers->date, vers->ts_conflict, CVSroot_directory, finfo->repository);

#ifdef SERVER_SUPPORT
            if (server_active)
            {
                /* We need to update the entries line on the client side.
                   It is possible we will later update it again via
                   server_updated or some such, but that is OK.  */
                server_update_entries
                  (finfo->file, finfo->update_dir, finfo->repository,
                   strcmp (vers->ts_rcs, vers->ts_user) == 0 ?
                   SERVER_UPDATED : SERVER_MERGED);
            }
#endif
        }
    }
}
```

A.8 client.c

```
/* CVS client-related stuff.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif /* HAVE_CONFIG_H */

#include <assert.h>
#include "cvs.h"
#include "getline.h"
#include "edit.h"
#include "buffer.h"

#ifdef CLIENT_SUPPORT

#include "md5.h"

#if defined(AUTH_CLIENT_SUPPORT) || HAVE_KERBEROS || defined(SOCK_ERRNO) || defined(SOCK_STRERROR)
#  ifdef HAVE_WINSOCK_H
#    include <winsock.h>
#  else /* No winsock.h */
#    include <sys/socket.h>
#    include <netinet/in.h>
#    include <netdb.h>
#  endif /* No winsock.h */
#endif

/* If SOCK_ERRNO is defined, then send()/recv() and other socket calls
   do not set errno, but that this macro should be used to obtain an
   error code.  This probably doesn't make sense unless
   NO_SOCKET_TO_FD is also defined. */
#ifndef SOCK_ERRNO
#define SOCK_ERRNO errno
#endif

/* If SOCK_STRERROR is defined, then the error codes returned by
   socket operations are not known to strerror, and this macro must be
   used instead to convert those error codes to strings. */
#ifndef SOCK_STRERROR
#  define SOCK_STRERROR strerror

#  if STDC_HEADERS
#    include <string.h>
#  endif

#  ifndef strerror
extern char *strerror ();
#  endif
#endif /* ! SOCK_STRERROR */

#if HAVE_KERBEROS
#define CVS_PORT 24011

#if HAVE_KERBEROS
#include <krb.h>

extern char *krb_realmofhost ();
#ifndef HAVE_KRB_GET_ERR_TEXT
#define krb_get_err_text(status) krb_err_txt[status]
#endif /* HAVE_KRB_GET_ERR_TEXT */

/* Information we need if we are going to use Kerberos encryption.  */
static C_Block kblock;
static Key_schedule sched;

#endif /* HAVE_KERBEROS */

#endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI

#ifdef HAVE_GSSAPI_H
#include <gssapi.h>
#endif
#ifdef HAVE_GSSAPI_GSSAPI_H
#include <gssapi/gssapi.h>
#endif
#ifdef HAVE_GSSAPI_GSSAPI_GENERIC_H
#include <gssapi/gssapi_generic.h>
```

```
#endif

#ifndef HAVE_GSS_C_NT_HOSTBASED_SERVICE
#define GSS_C_NT_HOSTBASED_SERVICE gss_nt_service_name
#endif

/* This is needed for GSSAPI encryption.  */
static gss_ctx_id_t gcontext;

static int connect_to_gserver PROTO((int, int, struct hostent *));
static int connect_to_pserver PROTO((int, int, struct hostent *));
static int connect_to_kserver PROTO((int, int, struct hostent *));

#endif /* HAVE_GSSAPI */

static void add_prune_candidate PROTO((char *));

void add_remote (char* file, char* server, char* root, char* repository);

/* All the commands.  */
int add PROTO((int argc, char **argv));
int admin PROTO((int argc, char **argv));
int checkout PROTO((int argc, char **argv));
int commit PROTO((int argc, char **argv));
int diff PROTO((int argc, char **argv));
int history PROTO((int argc, char **argv));
int import PROTO((int argc, char **argv));
int cvslog PROTO((int argc, char **argv));
int patch PROTO((int argc, char **argv));
int release PROTO((int argc, char **argv));
int cvsremove PROTO((int argc, char **argv));
int rtag PROTO((int argc, char **argv));
int status PROTO((int argc, char **argv));
int tag PROTO((int argc, char **argv));
int update PROTO((int argc, char **argv));

/* All the response handling functions.  */
static void handle_ok PROTO((char *, int));
static void handle_error PROTO((char *, int));
static void handle_valid_requests PROTO((char *, int));
static void handle_checked_in PROTO((char *, int));
static void handle_new_entry PROTO((char *, int));
static void handle_checksum PROTO((char *, int));
static void handle_copy_file PROTO((char *, int));
static void handle_updated PROTO((char *, int));
static void handle_merged PROTO((char *, int));
static void handle_patched PROTO((char *, int));
static void handle_rcs_diff PROTO((char *, int));
static void handle_removed PROTO((char *, int));
static void handle_remove_entry PROTO((char *, int));
static void handle_set_static_directory PROTO((char *, int));
static void handle_clear_static_directory PROTO((char *, int));
static void handle_set_sticky PROTO((char *, int));
static void handle_clear_sticky PROTO((char *, int));
static void handle_set_checkin_prog PROTO((char *, int));
static void handle_set_update_prog PROTO((char *, int));
static void handle_module_expansion PROTO((char *, int));
static void handle_wrapper_rcs_option PROTO((char *, int));
static void handle_m PROTO((char *, int));
static void handle_e PROTO((char *, int));
static void handle_f PROTO((char *, int));
static void handle_notified PROTO((char *, int));

void add_remote_tag (char* file, char* root, char* server, char* repository, char* revision);
static size_t try_read_from_server PROTO ((char *, size_t));
int fetching_remote = 0;
int adding_remote = 0;
#endif /* CLIENT_SUPPORT */

#if defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT)

/* Shared with server.  */

/*
 * Return a malloc'd, '\0'-terminated string
 * corresponding to the mode in SB.
 */
char *
#ifdef __STDC__
mode_to_string (mode_t mode)
#else /* ! __STDC__ */
mode_to_string (mode)
    mode_t mode;
#endif /* __STDC__ */
{
    char buf[18], u[4], g[4], o[4];
    int i;

    i = 0;
    if (mode & S_IRUSR) u[i++] = 'r';
```

```
    if (mode & S_IWUSR) u[i++] = 'w';
    if (mode & S_IXUSR) u[i++] = 'x';
    u[i] = '\0';

    i = 0;
    if (mode & S_IRGRP) g[i++] = 'r';
    if (mode & S_IWGRP) g[i++] = 'w';
    if (mode & S_IXGRP) g[i++] = 'x';
    g[i] = '\0';

    i = 0;
    if (mode & S_IROTH) o[i++] = 'r';
    if (mode & S_IWOTH) o[i++] = 'w';
    if (mode & S_IXOTH) o[i++] = 'x';
    o[i] = '\0';

    sprintf(buf, "u=%s,g=%s,o=%s", u, g, o);
    return xstrdup(buf);
}

/*
 * Change mode of FILENAME to MODE_STRING.
 * Returns 0 for success or errno code.
 * If RESPECT_UMASK is set, then honor the umask.
 */
int
change_mode (filename, mode_string, respect_umask)
    char *filename;
    char *mode_string;
    int respect_umask;
{
#ifdef CHMOD_BROKEN
    char *p;
    int writeable = 0;

    /* We can only distinguish between
         1) readable
         2) writeable
         3) Picasso's "Blue Period"
       We handle the first two. */
    p = mode_string;
    while (*p != '\0')
    {
        if ((p[0] == 'u' || p[0] == 'g' || p[0] == 'o') && p[1] == '=')
        {
            char *q = p + 2;
            while (*q != ',' && *q != '\0')
            {
                if (*q == 'w')
                    writeable = 1;
                ++q;
            }
        }
        /* Skip to the next field.  */
        while (*p != ',' && *p != '\0')
            ++p;
        if (*p == ',')
            ++p;
    }

    /* xchmod honors the umask for us.  In the !respect_umask case, we
       don't try to cope with it (probably to handle that well, the server
       needs to deal with modes in data structures, rather than via the
       modes in temporary files).  */
    xchmod (filename, writeable);
        return 0;

#else /* ! CHMOD_BROKEN */

    char *p;
    mode_t mode = 0;
    mode_t oumask;

    p = mode_string;
    while (*p != '\0')
    {
        if ((p[0] == 'u' || p[0] == 'g' || p[0] == 'o') && p[1] == '=')
        {
            int can_read = 0, can_write = 0, can_execute = 0;
            char *q = p + 2;
            while (*q != ',' && *q != '\0')
            {
                if (*q == 'r')
                    can_read = 1;
                else if (*q == 'w')
                    can_write = 1;
                else if (*q == 'x')
                    can_execute = 1;
                ++q;
            }
```

```
            if (p[0] == 'u')
            {
                if (can_read)
                    mode |= S_IRUSR;
                if (can_write)
                    mode |= S_IWUSR;
                if (can_execute)
                    mode |= S_IXUSR;
            }
            else if (p[0] == 'g')
            {
                if (can_read)
                    mode |= S_IRGRP;
                if (can_write)
                    mode |= S_IWGRP;
                if (can_execute)
                    mode |= S_IXGRP;
            }
            else if (p[0] == 'o')
            {
                if (can_read)
                    mode |= S_IROTH;
                if (can_write)
                    mode |= S_IWOTH;
                if (can_execute)
                    mode |= S_IXOTH;
            }
        }
        /* Skip to the next field.  */
        while (*p != ',' && *p != '\0')
            ++p;
        if (*p == ',')
            ++p;
    }

    if (respect_umask)
    {
        oumask = umask (0);
        (void) umask (oumask);
        mode &= ~oumask;
    }

    if (chmod (filename, mode) < 0)
        return errno;
    return 0;
#endif /* ! CHMOD_BROKEN */
}

#endif /* CLIENT_SUPPORT or SERVER_SUPPORT */

#ifdef CLIENT_SUPPORT

int client_prune_dirs;

static List *ignlist = (List *) NULL;

/* Buffer to write to the server.  */
static struct buffer *to_server;
/* The stream underlying to_server, if we are using a stream.  */
static FILE *to_server_fp;

/* Buffer used to read from the server.  */
static struct buffer *from_server;
/* The stream underlying from_server, if we are using a stream.  */
static FILE *from_server_fp;

/* Process ID of rsh subprocess.  */
static int rsh_pid = -1;


/* We want to be able to log data sent between us and the server.  We
   do it using log buffers.  Each log buffer has another buffer which
   handles the actual I/O, and a file to log information to.

   This structure is the closure field of a log buffer.  */

struct log_buffer
{
    /* The underlying buffer.  */
    struct buffer *buf;
    /* The file to log information to.  */
    FILE *log;
};

static struct buffer *log_buffer_initialize
  PROTO((struct buffer *, FILE *, int, void (*) (struct buffer *)));
static int log_buffer_input PROTO((void *, char *, int, int, int *));
static int log_buffer_output PROTO((void *, const char *, int, int *));
static int log_buffer_flush PROTO((void *));
static int log_buffer_block PROTO((void *, int));
```

```
static int log_buffer_shutdown PROTO((void *));

/* Create a log buffer.  */

static struct buffer *
log_buffer_initialize (buf, fp, input, memory)
     struct buffer *buf;
     FILE *fp;
     int input;
     void (*memory) PROTO((struct buffer *));
{
  struct log_buffer *n;

  n = (struct log_buffer *) xmalloc (sizeof *n);
  n->buf = buf;
  n->log = fp;
  return buf_initialize (input ? log_buffer_input : NULL,
                         input ? NULL : log_buffer_output,
                         input ? NULL : log_buffer_flush,
                         log_buffer_block,
                         log_buffer_shutdown,
                         memory,
                         n);
}

/* The input function for a log buffer.  */

static int
log_buffer_input (closure, data, need, size, got)
     void *closure;
     char *data;
     int need;
     int size;
     int *got;
{
  struct log_buffer *lb = (struct log_buffer *) closure;
  int status;
  size_t n_to_write;

  if (lb->buf->input == NULL)
    abort ();

  status = (*lb->buf->input) (lb->buf->closure, data, need, size, got);
  if (status != 0)
    return status;

  if (*got > 0)
  {
    n_to_write = *got;
    if (fwrite (data, 1, n_to_write, lb->log) != n_to_write)
      error (0, errno, "writing to log file");
  }

  return 0;
}

/* The output function for a log buffer.  */

static int
log_buffer_output (closure, data, have, wrote)
     void *closure;
     const char *data;
     int have;
     int *wrote;
{
  struct log_buffer *lb = (struct log_buffer *) closure;
  int status;
  size_t n_to_write;

  if (lb->buf->output == NULL)
    abort ();

  status = (*lb->buf->output) (lb->buf->closure, data, have, wrote);
  if (status != 0)
    return status;

  if (*wrote > 0)
  {
    n_to_write = *wrote;
    if (fwrite (data, 1, n_to_write, lb->log) != n_to_write)
      error (0, errno, "writing to log file");
  }

  return 0;
}

/* The flush function for a log buffer.  */

static int
log_buffer_flush (closure)
```

```
    void *closure;
{
  struct log_buffer *lb = (struct log_buffer *) closure;

  if (lb->buf->flush == NULL)
    abort ();

  /* We don't really have to flush the log file here, but doing it
     will let tail -f on the log file show what is sent to the
     network as it is sent.  */
  if (fflush (lb->log) != 0)
    error (0, errno, "flushing log file");

  return (*lb->buf->flush) (lb->buf->closure);
}

/* The block function for a log buffer.  */

static int
log_buffer_block (closure, block)
    void *closure;
    int block;
{
  struct log_buffer *lb = (struct log_buffer *) closure;

  if (block)
    return set_block (lb->buf);
  else
    return set_nonblock (lb->buf);
}

/* The shutdown function for a log buffer.  */

static int
log_buffer_shutdown (closure)
    void *closure;
{
  struct log_buffer *lb = (struct log_buffer *) closure;
  int retval;

  retval = buf_shutdown (lb->buf);
  if (fclose (lb->log) < 0)
    error (0, errno, "closing log file");
  return retval;
}

#ifdef NO_SOCKET_TO_FD

/* Under certain circumstances, we must communicate with the server
   via a socket using send() and recv().  This is because under some
   operating systems (OS/2 and Windows 95 come to mind), a socket
   cannot be converted to a file descriptor -- it must be treated as a
   socket and nothing else.

   We may also need to deal with socket routine error codes differently
   in these cases.  This is handled through the SOCK_ERRNO and
   SOCK_STRERROR macros. */

static int use_socket_style = 0;
static int server_sock;

/* These routines implement a buffer structure which uses send and
   recv.  The buffer is always in blocking mode so we don't implement
   the block routine.  */

/* Note that it is important that these routines always handle errors
   internally and never return a positive errno code, since it would in
   general be impossible for the caller to know in general whether any
   error code came from a socket routine (to decide whether to use
   SOCK_STRERROR or simply strerror to print an error message). */

/* We use an instance of this structure as the closure field.  */

struct socket_buffer
{
  /* The socket number.  */
  int socket;
};

static struct buffer *socket_buffer_initialize
  PROTO ((int, int, void (*) (struct buffer *)));
static int socket_buffer_input PROTO((void *, char *, int, int, int *));
static int socket_buffer_output PROTO((void *, const char *, int, int *));
static int socket_buffer_flush PROTO((void *));

/* Create a buffer based on a socket.  */

static struct buffer *
socket_buffer_initialize (socket, input, memory)
    int socket;
```

```
    int input;
    void (*memory) PROTO((struct buffer *));
{
  struct socket_buffer *n;

  n = (struct socket_buffer *) xmalloc (sizeof *n);
  n->socket = socket;
  return buf_initialize (input ? socket_buffer_input : NULL,
                         input ? NULL : socket_buffer_output,
                         input ? NULL : socket_buffer_flush,
                         (int (*) PROTO((void *, int))) NULL,
                         (int (*) PROTO((void *))) NULL,
                         memory,
                         n);
}

/* The buffer input function for a buffer built on a socket.  */

static int
socket_buffer_input (closure, data, need, size, got)
    void *closure;
    char *data;
    int need;
    int size;
    int *got;
{
  struct socket_buffer *sb = (struct socket_buffer *) closure;
  int nbytes;

  /* I believe that the recv function gives us exactly the semantics
     we want.  If there is a message, it returns immediately with
     whatever it could get.  If there is no message, it waits until
     one comes in.  In other words, it is not like read, which in
     blocking mode normally waits until all the requested data is
     available.  */

  *got = 0;

  do
  {

    /* Note that for certain (broken?) networking stacks, like
       VMS's UCX (not sure what version, problem reported with
       recv() in 1997), and (according to windows-NT/config.h)
       Windows NT 3.51, we must call recv or send with a
       moderately sized buffer (say, less than 200K or something),
       or else there may be network errors (somewhat hard to
       produce, e.g. WAN not LAN or some such).  buf_read_data
       makes sure that we only recv() BUFFER_DATA_SIZE bytes at
       a time.  */

    nbytes = recv (sb->socket, data, size, 0);
    if (nbytes < 0)
      error (1, 0, "reading from server: %s", SOCK_STRERROR (SOCK_ERRNO));
    if (nbytes == 0)
    {
      /* End of file (for example, the server has closed
         the connection).  If we've already read something, we
         just tell the caller about the data, not about the end of
         file.  If we've read nothing, we return end of file.  */
      if (*got == 0)
        return -1;
      else
        return 0;
    }
    need -= nbytes;
    size -= nbytes;
    data += nbytes;
    *got += nbytes;
  }
  while (need > 0);

  return 0;
}

/* The buffer output function for a buffer built on a socket.  */

static int
socket_buffer_output (closure, data, have, wrote)
    void *closure;
    const char *data;
    int have;
    int *wrote;
{
  struct socket_buffer *sb = (struct socket_buffer *) closure;

  *wrote = have;

  /* See comment in socket_buffer_input regarding buffer size we pass
     to send and recv.  */
```

```
#ifdef SEND_NEVER_PARTIAL
    /* If send() never will produce a partial write, then just do it.  This
       is needed for systems where its return value is something other than
       the number of bytes written.  */
    if (send (sb->socket, data, have, 0) < 0)
        error (1, 0, "writing to server socket: %s", SOCK_STRERROR (SOCK_ERRNO));
#else
    while (have > 0)
    {
        int nbytes;

        nbytes = send (sb->socket, data, have, 0);
        if (nbytes < 0)
            error (1, 0, "writing to server socket: %s", SOCK_STRERROR (SOCK_ERRNO));

        have -= nbytes;
        data += nbytes;
    }
#endif

    return 0;
}

/* The buffer flush function for a buffer built on a socket.  */

/*ARGSUSED*/
static int
socket_buffer_flush (closure)
    void *closure;
{
    /* Nothing to do.  Sockets are always flushed.  */
    return 0;
}

#endif /* NO_SOCKET_TO_FD */

/*
 * Read a line from the server.  Result does not include the terminating \n.
 *
 * Space for the result is malloc'd and should be freed by the caller.
 *
 * Returns number of bytes read.
 */
static int
read_line (resultp)
    char **resultp;
{
    int status;
    char *result;
    int len;

    status = buf_flush (to_server, 1);
    if (status != 0)
        error (1, status, "writing to server");

    status = buf_read_line (from_server, &result, &len);
    if (status != 0)
    {
        if (status == -1)
            error (1, 0, "end of file from server (consult above messages if any)");
        else if (status == -2)
            error (1, 0, "out of memory");
        else
            error (1, status, "reading from server");
    }

    if (resultp != NULL)
        *resultp = result;
    else
        free (result);

    return len;
}

#endif /* CLIENT_SUPPORT */


#if defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT)

/*
 * Zero if compression isn't supported or requested; non-zero to indicate
 * a compression level to request from gzip.
 */
int gzip_level;

/*
 * Level of compression to use when running gzip on a single file.
 */
int file_gzip_level;
```

```
int filter_through_gzip (fd, dir, level, pidp)
    int fd, dir, level;
    pid_t *pidp;
{
    static char buf[5] = "-";
    static char *gzip_argv[3] = { "gzip", buf };

    sprintf (buf+1, "%d", level);
    return filter_stream_through_program (fd, dir, &gzip_argv[0], pidp);
}

int filter_through_gunzip (fd, dir, pidp)
    int fd, dir;
    pid_t *pidp;
{
    static char *gunzip_argv[3] = { "gzip", "-d" };
    return filter_stream_through_program (fd, dir, &gunzip_argv[0], pidp);
}

#endif /* CLIENT_SUPPORT or SERVER_SUPPORT */

#ifdef CLIENT_SUPPORT

/*
 * The Repository for the top level of this command (not necessarily
 * the CVSROOT, just the current directory at the time we do it).
 */
static char *toplevel_repos;

/* Working directory when we first started.  Note: we could speed things
   up on some systems by using savecwd.h here instead of just always
   storing a name.  */
char *toplevel_wd;

static void
handle_ok (args, len)
    char *args;
    int len;
{
    return;
}

static void
handle_error (args, len)
    char *args;
    int len;
{
    int something_printed;

    /*
     * First there is a symbolic error code followed by a space, which
     * we ignore.
     */
    char *p = strchr (args, ' ');
    if (p == NULL)
    {
        error (0, 0, "invalid data from cvs server");
        return;
    }
    ++p;
    len -= p - args;
    something_printed = 0;
    for (; len > 0; --len)
    {
        something_printed = 1;
        putc (*p++, stderr);
    }
    if (something_printed)
        putc ('\n', stderr);
}

static void
handle_valid_requests (args, len)
    char *args;
    int len;
{
    char *p = args;
    char *q;
    struct request *rq;
    do
    {
        q = strchr (p, ' ');
        if (q != NULL)
            *q++ = '\0';
        for (rq = requests; rq->name != NULL; ++rq)
        {
            if (strcmp (rq->name, p) == 0)
                break;
        }
```

```
        if (rq->name == NULL)
            /*
             * It is a request we have never heard of (and thus never
             * will want to use).  So don't worry about it.
             */
            ;
        else
        {
            if (rq->status == rq_enableme)
            {
                /*
                 * Server wants to know if we have this, to enable the
                 * feature.
                 */
                send_to_server (rq->name, 0);
                send_to_server ("\012", 0);
            }
            else
                rq->status = rq_supported;
        }
        p = q;
    } while (q != NULL);
    for (rq = requests; rq->name != NULL; ++rq)
    {
        if (rq->status == rq_essential)
            error (1, 0, "request `%s' not supported by server", rq->name);
        else if (rq->status == rq_optional)
            rq->status = rq_not_supported;
    }
}

/* This variable holds the result of Entries_Open, so that we can
   close Entries_Close on it when we move on to a new directory, or
   when we finish.  */
static List *last_entries;

/*
 * Do all the processing for PATHNAME, where pathname consists of the
 * repository and the filename.  The parameters we pass to FUNC are:
 * DATA is just the DATA parameter which was passed to
 * call_in_directory; ENT_LIST is a pointer to an entries list (which
 * we manage the storage for); SHORT_PATHNAME is the pathname of the
 * file relative to the (overall) directory in which the command is
 * taking place; and FILENAME is the filename portion only of
 * SHORT_PATHNAME.  When we call FUNC, the curent directory points to
 * the directory portion of SHORT_PATHNAME.  */

static char *last_dir_name;

static void
call_in_directory (pathname, func, data)
    char *pathname;
    void (*func) PROTO((char *data, List *ent_list, char *short_pathname,
                        char *filename));
    char *data;
{
    char *dir_name;
    char *filename;
    /* This is what we get when we hook up the directory (working directory
       name) from PATHNAME with the filename from REPOSNAME.  For example:
       pathname: ccvs/src/
       reposname: /u/src/master/ccvs/foo/ChangeLog
       short_pathname: ccvs/src/ChangeLog
       */
    char *short_pathname;
    char *p;

    /*
     * Do the whole descent in parallel for the repositories, so we
     * know what to put in CVS/Repository files.  I'm not sure the
     * full hair is necessary since the server does a similar
     * computation; I suspect that we only end up creating one
     * directory at a time anyway.
     *
     * Also note that we must *only* worry about this stuff when we
     * are creating directories; 'cvs co foo/bar; cd foo/bar; cvs co
     * CVSROOT; cvs update' is legitimate, but in this case
     * foo/bar/CVSROOT/CVS/Repository is not a subdirectory of
     * foo/bar/CVS/Repository.
     */
    char *reposname;
    char *short_repos;
    char *reposdirname;
    char *rdirp;
    int reposdirname_absolute;

    reposname = NULL;
    read_line (&reposname);
    assert (reposname != NULL);
```

```
    reposdirname_absolute = 0;
900 if (strncmp (reposname, toplevel_repos, strlen (toplevel_repos)) != 0)
    {
        reposdirname_absolute = 1;
        short_repos = reposname;
    }
    else
    {
        short_repos = reposname + strlen (toplevel_repos) + 1;
        if (short_repos[-1] != '/')
        {
910         reposdirname_absolute = 1;
            short_repos = reposname;
        }
    }
    reposdirname = xstrdup (short_repos);
    p = strrchr (reposdirname, '/');
    if (p == NULL)
    {
        reposdirname = xrealloc (reposdirname, 2);
        reposdirname[0] = '.'; reposdirname[1] = '\0';
920 }
    else
        *p = '\0';

    dir_name = xstrdup (pathname);
    p = strrchr (dir_name, '/');
    if (p == NULL)
    {
        dir_name = xrealloc (dir_name, 2);
        dir_name[0] = '.'; dir_name[1] = '\0';
930 }
    else
        *p = '\0';
    if (client_prune_dirs)
        add_prune_candidate (dir_name);

    filename = strrchr (short_repos, '/');
    if (filename == NULL)
        filename = short_repos;
    else
940     ++filename;

    short_pathname = xmalloc (strlen (pathname) + strlen (filename) + 5);
    strcpy (short_pathname, pathname);
    strcat (short_pathname, filename);

    if (last_dir_name == NULL
        || strcmp (last_dir_name, dir_name) != 0)
    {
        int newdir;
950

        if (strcmp (command_name, "export") != 0)
            if (last_entries)
                Entries_Close (last_entries);

        if (last_dir_name)
            free (last_dir_name);
        last_dir_name = dir_name;

        if (toplevel_wd == NULL)
960     {
            toplevel_wd = xgetwd ();
            if (toplevel_wd == NULL)
                error (1, errno, "could not get working directory");
        }

        if (CVS_CHDIR (toplevel_wd) < 0)
            error (1, errno, "could not chdir to %s", toplevel_wd);
        newdir = 0;

970     /* Create the CVS directory at the top level if needed.  The
           isdir seems like an unneeded system call, but it *does*
           need to be called both if the CVS_CHDIR below succeeds
           (e.g.  "cvs co .") or if it fails (e.g. basicb-1a in
           testsuite).  We only need to do this for the "." case,
           since the server takes care of forcing this directory to be
           created in all other cases.  If we don't create CVSADM
           here, the call to Entries_Open below will fail.  FIXME:
           perhaps this means that we should change our algorithm
           below that calls Create_Admin instead of having this code
980        here? */
        if (/* I think the reposdirname_absolute case has to do with
               things like "cvs update /foo/bar".  In any event, the
               code below which tries to put toplevel_repos into
               CVS/Repository is almost surely unsuited to
               the reposdirname_absolute case.  */
            !reposdirname_absolute
            && (strcmp (dir_name, ".") == 0)
            && ! isdir (CVSADM))
```

```
    {
990     char *repo;
        char *r;

        newdir = 1;

        repo = xmalloc (strlen (toplevel_repos)
                        + 10);
        strcpy (repo, toplevel_repos);
        r = repo + strlen (repo);
        if (r[-1] != '.' || r[-2] != '/')
1000        strcpy (r, "/.");

        Create_Admin (".", ".", repo, (char *) NULL,
                      (char *) NULL, 0, 1);

        free (repo);
    }

    if ( CVS_CHDIR (dir_name) < 0)
    {
1010    char *dir;
        char *dirp;

        if (! existence_error (errno))
            error (1, errno, "could not chdir to %s", dir_name);

        /* Directory does not exist, we need to create it.  */
        newdir = 1;

        /* Provided we are willing to assume that directories get
1020       created one at a time, we could simplify this a lot.
           Do note that one aspect still would need to walk the
           dir_name path: the checking for "fncmp (dir, CVSADM)".  */

        dir = xmalloc (strlen (dir_name) + 1);
        dirp = dir_name;
        rdirp = reposdirname;

        /* This algorithm makes nested directories one at a time
           and create CVS administration files in them.  For
1030       example, we're checking out foo/bar/baz from the
           repository:

           1) create foo, point CVS/Repository to <root>/foo
           2)     .. foo/bar                  .. <root>/foo/bar
           3)     .. foo/bar/baz               .. <root>/foo/bar/baz

           As you can see, we're just stepping along DIR_NAME (with
           DIRP) and REPOSDIRNAME (with RDIRP) respectively.

1040       We need to be careful when we are checking out a
           module, however, since DIR_NAME and REPOSDIRNAME are not
           going to be the same.  Since modules will not have any
           slashes in their names, we should watch the output of
           STRCHR to decide whether or not we should use STRCHR on
           the RDIRP.  That is, if we're down to a module name,
           don't keep picking apart the repository directory name.  */

        do
        {
1050        dirp = strchr (dirp, '/');
            if (dirp)
            {
                strncpy (dir, dir_name, dirp - dir_name);
                dir[dirp - dir_name] = '\0';
                /* Skip the slash.  */
                ++dirp;
                if (rdirp == NULL)
                    /* This just means that the repository string has
                       fewer components than the dir_name string.  But
1060                   that is OK (e.g. see modules3-8 in testsuite).  */
                    ;
                else
                    rdirp = strchr (rdirp, '/');
            }
            else
            {
                /* If there are no more slashes in the dir name,
                   we're down to the most nested directory -OR- to
                   the name of a module.  In the first case, we
1070               should be down to a DIRP that has no slashes,
                   so it won't help/hurt to do another STRCHR call
                   on DIRP.  It will definitely hurt, however, if
                   we're down to a module name, since a module
                   name can point to a nested directory (that is,
                   DIRP will still have slashes in it.  Therefore,
                   we should set it to NULL so the routine below
                   copies the contents of REMOTEDIRNAME onto the
                   root repository directory (does this if rdirp
```

```
                    is set to NULL, because we used to do an extra
1080                STRCHR call here). */

                rdirp = NULL;
                strcpy (dir, dir_name);
            }

            if (fncmp (dir, CVSADM) == 0)
            {
                error (0, 0, "cannot create a directory named %s", dir);
                error (0, 0, "because CVS uses \"%s\" for its own uses",
1090                   CVSADM);
                error (1, 0, "rename the directory and try again");
            }

            if (mkdir_if_needed (dir))
            {
                /* It already existed, fine.  Just keep going.  */
            }
            else if (strcmp (command_name, "export") == 0)
                /* Don't create CVSADM directories if this is export.  */
1100            ;
            else
            {
                /*
                 * Put repository in CVS/Repository.  For historical
                 * (pre-CVS/Root) reasons, this is an absolute pathname,
                 * but what really matters is the part of it which is
                 * relative to cvsroot.
                 */
                char *repo;
1110            char *r, *b;

                repo = xmalloc (strlen (reposdirname)
                                + strlen (toplevel_repos)
                                + 80);
                if (reposdirname_absolute)
                    r = repo;
                else
                {
                    strcpy (repo, toplevel_repos);
1120                strcat (repo, "/");
                    r = repo + strlen (repo);
                }

                if (rdirp)
                {
                    /* See comment near start of function; the only
                       way that the server can put the right thing
                       in each CVS/Repository file is to create the
                       directories one at a time.  I think that the
1130                   CVS server has been doing this all along.  */
                    error (0, 0, "\
warning: server is not creating directories one at a time");
                    strncpy (r, reposdirname, rdirp - reposdirname);
                    r[rdirp - reposdirname] = '\0';
                }
                else
                    strcpy (r, reposdirname);

                Create_Admin (dir, dir, repo,
1140                          (char *)NULL, (char *)NULL, 0, 0);
                free (repo);

                b = strrchr (dir, '/');
                if (b == NULL)
                    Subdir_Register ((List *) NULL, (char *) NULL, dir);
                else
                {
                    *b = '\0';
                    Subdir_Register ((List *) NULL, dir, b + 1);
1150                *b = '/';
                }
            }

            if (rdirp != NULL)
            {
                /* Skip the slash.  */
                ++rdirp;
            }

1160        } while (dirp != NULL);
            free (dir);
            /* Now it better work.  */
            if ( CVS_CHDIR (dir_name) < 0)
                error (1, errno, "could not chdir to %s", dir_name);
        }

        if (strcmp (command_name, "export") != 0)
        {
```

```
            last_entries = Entries_Open (0, dir_name);

            /* If this is a newly created directory, we will record
               all subdirectory information, so call Subdirs_Known in
               case there are no subdirectories.  If this is not a
               newly created directory, it may be an old working
               directory from before we recorded subdirectory
               information in the Entries file.  We force a search for
               all subdirectories now, to make sure our subdirectory
               information is up to date.  If the Entries file does
               record subdirectory information, then this call only
               does list manipulation.  */
            if (newdir)
                Subdirs_Known (last_entries);
            else
            {
                List *dirlist;

                dirlist = Find_Directories ((char *) NULL, W_LOCAL,
                                            last_entries);
                dellist (&dirlist);
            }
        }
    }
    else
        free (dir_name);
    free (reposdirname);
    (*func) (data, last_entries, short_pathname, filename);
    free (short_pathname);
    free (reposname);
}

static void
copy_a_file (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    char *newname;
#ifdef USE_VMS_FILENAMES
    char *p;
#endif

    read_line (&newname);

#ifdef USE_VMS_FILENAMES
    /* Mogrify the filename so VMS is happy with it. */
    for(p = newname; *p; p++)
      if(*p == '.' || *p == '#') *p = '_';
#endif

    copy_file (filename, newname);
    free (newname);
}

static void
handle_copy_file (args, len)
    char *args;
    int len;
{
    call_in_directory (args, copy_a_file, (char *)NULL);
}


static void read_counted_file PROTO ((char *, char *));

/* Read from the server the count for the length of a file, then read
   the contents of that file and write them to FILENAME.  FULLNAME is
   the name of the file for use in error messages.  FIXME-someday:
   extend this to deal with compressed files and make update_entries
   use it.  On error, gives a fatal error.  */
static void
read_counted_file (filename, fullname)
    char *filename;
    char *fullname;
{
    char *size_string;
    size_t size;
    char *buf;

    /* Pointers in buf to the place to put data which will be read,
       and the data which needs to be written, respectively.  */
    char *pread;
    char *pwrite;
    /* Number of bytes left to read and number of bytes in buf waiting to
       be written, respectively.  */
    size_t nread;
    size_t nwrite;
```

```
    FILE *fp;

    read_line (&size_string);
    if (size_string[0] == 'z')
        error (1, 0, "\
protocol error: compressed files not supported for that operation");
    /* FIXME: should be doing more error checking, probably.  Like using
       strtoul and making sure we used up the whole line.  */
    size = atoi (size_string);
    free (size_string);

    /* A more sophisticated implementation would use only a limited amount
       of buffer space (8K perhaps), and read that much at a time.  We allocate
       a buffer for the whole file only to make it easy to keep track what
       needs to be read and written.  */
    buf = xmalloc (size);

    /* FIXME-someday: caller should pass in a flag saying whether it
       is binary or not.  I haven't carefully looked into whether
       CVS/Template files should use local text file conventions or
       not.  */
    fp = CVS_FOPEN (filename, "wb");
    if (fp == NULL)
        error (1, errno, "cannot write %s", fullname);
    nread = size;
    nwrite = 0;
    pread = buf;
    pwrite = buf;
    while (nread > 0 || nwrite > 0)
    {
        size_t n;

        if (nread > 0)
        {
            n = try_read_from_server (pread, nread);
            nread -= n;
            pread += n;
            nwrite += n;
        }

        if (nwrite > 0)
        {
            n = fwrite (pwrite, 1, nwrite, fp);
            if (ferror (fp))
                error (1, errno, "cannot write %s", fullname);
            nwrite -= n;
            pwrite += n;
        }
    }
    free (buf);
    if (fclose (fp) < 0)
        error (1, errno, "cannot close %s", fullname);
}

/* OK, we want to swallow the "U foo.c" response and then output it only
   if we can update the file.  In the future we probably want some more
   systematic approach to parsing tagged text, but for now we keep it
   ad hoc.  "Why," I hear you cry, "do we not just look at the
   Update-existing and Created responses?"  That is an excellent question,
   and the answer is roughly conservatism/laziness--I haven't read through
   update.c enough to figure out the exact correspondence or lack thereof
   between those responses and a "U foo.c" line (note that Merged, from
   join_file, can be either "C foo" or "U foo" depending on the context).  */
/* Nonzero if we have seen +updated and not -updated.  */
static int updated_seen;
/* Filename from an "fname" tagged response within +updated/-updated.  */
static char *updated_fname;


/* Nonzero if we should arrange to return with a failure exit status.  */
static int failure_exit;

/*
 * The time stamp of the last file we registered.
 */
static time_t last_register_time;

/*
 * The Checksum response gives the checksum for the file transferred
 * over by the next Updated, Merged or Patch response.  We just store
 * it here, and then check it in update_entries.
 */

static int stored_checksum_valid;
static unsigned char stored_checksum[16];

static void
handle_checksum (args, len)
    char *args;
```

```
    int len;
{
    char *s;
    char buf[3];
    int i;

    if (stored_checksum_valid)
        error (1, 0, "Checksum received before last one was used");

    s = args;
    buf[2] = '\0';
    for (i = 0; i < 16; i++)
    {
        char *bufend;

        buf[0] = *s++;
        buf[1] = *s++;
        stored_checksum[i] = (char) strtol (buf, &bufend, 16);
        if (bufend != buf + 2)
            break;
    }

    if (i < 16 || *s != '\0')
        error (1, 0, "Invalid Checksum response: `%s'", args);

    stored_checksum_valid = 1;
}

static int stored_mode_valid;
static char *stored_mode;

static void handle_mode PROTO ((char *, int));

static void
handle_mode (args, len)
    char *args;
    int len;
{
    if (stored_mode_valid)
        error (1, 0, "protocol error: duplicate Mode");
    if (stored_mode != NULL)
        free (stored_mode);
    stored_mode = xstrdup (args);
    stored_mode_valid = 1;
}

/* Nonzero if time was specified in Mod-time.  */
static int stored_modtime_valid;
/* Time specified in Mod-time.  */
static time_t stored_modtime;

static void handle_mod_time PROTO ((char *, int));

static void
handle_mod_time (args, len)
    char *args;
    int len;
{
    if (stored_modtime_valid)
        error (0, 0, "protocol error: duplicate Mod-time");
    stored_modtime = get_date (args, NULL);
    if (stored_modtime == (time_t) -1)
        error (0, 0, "protocol error: cannot parse date %s", args);
    else
        stored_modtime_valid = 1;
}

/*
 * If we receive a patch, but the patch program fails to apply it, we
 * want to request the original file.  We keep a list of files whose
 * patches have failed.
 */

char **failed_patches;
int failed_patches_count;

struct update_entries_data
{
    enum {
      /*
       * We are just getting an Entries line; the local file is
       * correct.
       */
      UPDATE_ENTRIES_CHECKIN,
      /* We are getting the file contents as well.  */
      UPDATE_ENTRIES_UPDATE,
      /*
       * We are getting a patch against the existing local file, not
       * an entire new file.
       */
```

```
        UPDATE_ENTRIES_PATCH,
        /*
         * We are getting an RCS change text (diff -n output) against
         * the existing local file, not an entire new file.
         */
        UPDATE_ENTRIES_RCS_DIFF
      } contents;

      enum {
          /* We are replacing an existing file.  */
          UPDATE_ENTRIES_EXISTING,
          /* We are creating a new file.  */
          UPDATE_ENTRIES_NEW,
          /* We don't know whether it is existing or new.  */
          UPDATE_ENTRIES_EXISTING_OR_NEW
      } existp;

      /*
       * String to put in the timestamp field or NULL to use the timestamp
       * of the file.
       */
      char *timestamp;
};

/* Update the Entries line for this file.  */
static void
update_entries (data_arg, ent_list, short_pathname, filename)
      char *data_arg;
      List *ent_list;
      char *short_pathname;
      char *filename;
{
      char *entries_line;
      struct update_entries_data *data = (struct update_entries_data *)data_arg;

      char* remote_file;
      char *cp;
      char *user;
      char *vn;
      /* Timestamp field.  Always empty according to the protocol.  */
      char *ts;
      char *options = NULL;
      char *tag = NULL;
      char *date = NULL;
      char *tag_or_date;
      char *scratch_entries = NULL;
      char *repository = NULL;
      int bin;

#ifdef UTIME_EXPECTS_WRITABLE
      int change_it_back = 0;
#endif

      read_line (&entries_line);

      /*
       * Parse the entries line.
       */
      scratch_entries = xstrdup (entries_line);

      if (scratch_entries[0] != '/')
          error (1, 0, "bad entries line `%s' from server", entries_line);
      user = scratch_entries + 1;
      if ((cp = strchr (user, '/')) == NULL)
          error (1, 0, "bad entries line `%s' from server", entries_line);
      *cp++ = '\0';
      vn = cp;
      if ((cp = strchr (vn, '/')) == NULL)
          error (1, 0, "bad entries line `%s' from server", entries_line);
      *cp++ = '\0';

      ts = cp;
      if ((cp = strchr (ts, '/')) == NULL)
          error (1, 0, "bad entries line `%s' from server", entries_line);
      *cp++ = '\0';

      options = cp;
      if ((cp = strchr (options, '/')) == NULL)
          error (1, 0, "bad entries line `%s' from server", entries_line);
      *cp++ = '\0';

      tag_or_date = cp;
      if ((cp = strchr (tag_or_date, '/')) == NULL)
        error (1, 0, "bad entried line `%s' from server", entries_line);
      *cp++ = '\0';

      /* Note that the root doesn't come from the server, because the server
         has non idea it's running as a server - it thinks it's running
         in a filesystem. That's hard to fix, so I won't. */
```

```
    repository = cp;

    /* If a slash ends the tag_or_date, ignore everything after it.  */
    cp = strchr (tag_or_date, '/');
    if (cp != NULL)
        *cp = '\0';
    if (*tag_or_date == 'T')
        tag = tag_or_date + 1;
    else if (*tag_or_date == 'D')
        date = tag_or_date + 1;

    /* Done parsing the entries line. */

    if (data->contents == UPDATE_ENTRIES_UPDATE
        || data->contents == UPDATE_ENTRIES_PATCH
        || data->contents == UPDATE_ENTRIES_RCS_DIFF)
    {
        char *size_string;
        char *mode_string;
        int size;
        char *buf;
        char *temp_filename;
        int use_gzip;
        int patch_failed;

        read_line (&mode_string);

        read_line (&size_string);
        if (size_string[0] == 'z')
        {
            use_gzip = 1;
            size = atoi (size_string+1);
        }
        else
        {
            use_gzip = 0;
            size = atoi (size_string);
        }
        free (size_string);

        /* Note that checking this separately from writing the file is
           a race condition: if the existing or lack thereof of the
           file changes between now and the actually calls which
           operate on it, we lose.  However (a) there are so many
           cases, I'm reluctant to try to fix them all, (b) in some
           cases the system might not even have a system call which
           does the right thing, and (c) it isn't clear this needs to
           work.  */
        if (data->existp == UPDATE_ENTRIES_EXISTING
            && !isfile (filename))
            /* Emit a warning and update the file anyway.  */
            error (0, 0, "warning: %s unexpectedly disappeared",
                   short_pathname);

        if (data->existp == UPDATE_ENTRIES_NEW
            && isfile (filename))
        {
            /* Emit a warning and refuse to update the file; we don't want
               to clobber a user's file.  */
            size_t nread;
            size_t toread;

            /* size should be unsigned, but until we get around to fixing
               that, work around it.  */
            size_t usize;

            char buf[8192];

            /* This error might be confusing; it isn't really clear to
               the user what to do about it.  Keep in mind that it has
               several causes: (1) something/someone creates the file
               during the time that CVS is running, (2) the repository
               has two files whose names clash for the client because
               of case-insensitivity or similar causes, (3) a special
               case of this is that a file gets renamed for example
               from a.c to A.C.  A "cvs update" on a case-insensitive
               client will get this error.  Repeating the update takes
               care of the problem, but is it clear to the user what
               is going on and what to do about it?, (4) the client
               has a file which the server doesn't know about (e.g. "?
               foo" file), and that name clashes with a file the
               server does know about, (5) classify.c will print the same
               message for other reasons.

               I hope the above paragraph makes it clear that making this
               clearer is not a one-line fix.  */
            error (0, 0, "move away %s; it is in the way", short_pathname);
            if (updated_fname != NULL)
            {
                cvs_output ("C ", 0);
```

```
                cvs_output (updated_fname, 0);
                cvs_output ("\n", 1);
            }
            failure_exit = 1;

        discard_file_and_return:
            /* Now read and discard the file contents.  */
            usize = size;
            nread = 0;
            while (nread < usize)
            {
                toread = usize - nread;
                if (toread > sizeof buf)
                    toread = sizeof buf;

                nread += try_read_from_server (buf, toread);
                if (nread == usize)
                    break;
            }

            free (mode_string);
            free (entries_line);

            /* The Mode, Mod-time, and Checksum responses should not carry
               over to a subsequent Created (or whatever) response, even
               in the error case.  */
            stored_mode_valid = 0;
            if (stored_mode != NULL)
                free (stored_mode);
            stored_modtime_valid = 0;
            stored_checksum_valid = 0;

            if (updated_fname != NULL)
            {
                free (updated_fname);
                updated_fname = NULL;
            }
            return;
        }

        temp_filename = xmalloc (strlen (filename) + 80);
#ifdef USE_VMS_FILENAMES
        /* A VMS rename of "blah.dat" to "foo" to implies a
           destination of "foo.dat" which is unfortinate for CVS */
        sprintf (temp_filename, "%s_new_", filename);
#else
#ifdef _POSIX_NO_TRUNC
        sprintf (temp_filename, ".new.%.9s", filename);
#else /* _POSIX_NO_TRUNC */
        sprintf (temp_filename, ".new.%s", filename);
#endif /* _POSIX_NO_TRUNC */
#endif /* USE_VMS_FILENAMES */

        buf = xmalloc (size);

        /* Some systems, like OS/2 and Windows NT, end lines with CRLF
           instead of just LF.  Format translation is done in the C
           library I/O funtions.  Here we tell them whether or not to
           convert -- if this file is marked "binary" with the RCS -kb
           flag, then we don't want to convert, else we do (because
           CVS assumes text files by default).  */

        if (options)
            bin = !(strcmp (options, "-kb"));
        else
            bin = 0;

        if (data->contents == UPDATE_ENTRIES_RCS_DIFF)
        {
            /* This is an RCS change text.  We just hold the change
               text in memory.  */

            if (use_gzip)
                error (1, 0,
                       "server error: gzip invalid with RCS change text");

            read_from_server (buf, size);
        }
        else
        {
            int fd;

            fd = CVS_OPEN (temp_filename,
                           (O_WRONLY | O_CREAT | O_TRUNC
                            | (bin ? OPEN_BINARY : 0)),
                           0777);

            if (fd < 0)
            {
                /* I can see a case for making this a fatal error; for
```

```
                a condition like disk full or network unreachable
                (for a file server), carrying on and giving an
                error on each file seems unnecessary.  But if it is
                a permission problem, or some such, then it is
                entirely possible that future files will not have
                the same problem.  */
             error (0, errno, "cannot write %s", short_pathname);
             goto discard_file_and_return;
          }

          if (size > 0)
          {
             read_from_server (buf, size);

             if (use_gzip)
                gunzip_and_write (fd, short_pathname, buf, size);
             else if (write (fd, buf, size) != size)
                error (1, errno, "writing %s", short_pathname);
          }

          if (close (fd) < 0)
             error (1, errno, "writing %s", short_pathname);
       }

       /* This is after we have read the file from the net (a change
          from previous versions, where the server would send us
          "M U foo.c" before Update-existing or whatever), but before
          we finish writing the file (arguably a bug).  The timing
          affects a user who wants status info about how far we have
          gotten, and also affects whether "U foo.c" appears in addition
          to various error messages.  */
       if (updated_fname != NULL)
       {
          cvs_output ("U ", 0);
          cvs_output (updated_fname, 0);
          cvs_output ("\n", 1);
          free (updated_fname);
          updated_fname = 0;
       }

       patch_failed = 0;

       if (data->contents == UPDATE_ENTRIES_UPDATE)
       {
         /* If we are fetching a remote file, then we put its contents
            in a temporary file inside the CVS admin directory, instead of
            blasting it over the real file in the working directory

            temp file format is CVS/remote_rev_name
         */
         if (fetching_remote) {
           remote_file = xmalloc (strlen (filename) + strlen (CVSADM_REMOTE_TMP) + strlen (tag) + 10);
           sprintf (remote_file, "%s_%s_%s", CVSADM_REMOTE_TMP, tag, filename);
           rename_file (temp_filename, remote_file);
         } else {
           rename_file (temp_filename, filename);
       }
       }

       else if (data->contents == UPDATE_ENTRIES_PATCH)
       {
#ifdef DONT_USE_PATCH
          /* Hmm.  We support only Rcs-diff, and the server supports
             only Patched (or else it would have sent Rcs-diff instead).
             Fall back to transmitting entire files.  */
          patch_failed = 1;
#else /* Use patch.  */
          int retcode;
          char *backup;
          struct stat s;

          backup = xmalloc (strlen (filename) + 5);
          strcpy (backup, filename);
          strcat (backup, "~");
          (void) unlink_file (backup);
          if (!isfile (filename))
             error (1, 0, "patch original file %s does not exist",
                    short_pathname);
          if ( CVS_STAT (temp_filename, &s) < 0)
             error (1, errno, "can't stat patch file %s", temp_filename);
          if (s.st_size == 0)
             retcode = 0;
          else
          {
             /* This behavior (in which -b takes an argument) is
                supported by GNU patch 2.1.  Apparently POSIX.2
                specifies a -b option without an argument.  GNU
                patch 2.1.5 implements this and therefore won't
                work here.  GNU patch versions after 2.1.5 are said
                to have a kludge which checks if the last 4 args
```

```
                are '-b SUFFIX ORIGFILE PATCHFILE' and if so emit a
                warning (I think -s suppresses it), and then behave
                as CVS expects.

                Of course this is yet one more reason why in the long
                run we want Rcs-diff to replace Patched. */

              run_setup (PATCH_PROGRAM);
              run_arg ("-f");
              run_arg ("-s");
              run_arg ("-b");
              run_arg ("~");
              run_arg (filename);
              run_arg (temp_filename);
              retcode = run_exec (DEVNULL, RUN_TTY, RUN_TTY, RUN_NORMAL);
            }
            /* FIXME: should we really be silently ignoring errors? */
            (void) unlink_file (temp_filename);
            if (retcode == 0)
            {
                /* FIXME: should we really be silently ignoring errors? */
                (void) unlink_file (backup);
            }
            else
            {
                int old_errno = errno;
                char *path_tmp;

                if (isfile (backup))
                    rename_file (backup, filename);

                /* Get rid of the patch reject file. */
                path_tmp = xmalloc (strlen (filename) + 10);
                strcpy (path_tmp, filename);
                strcat (path_tmp, ".rej");
                /* FIXME: should we really be silently ignoring errors? */
                (void) unlink_file (path_tmp);
                free (path_tmp);

                error (retcode == -1 ? 1 : 0, retcode == -1 ? old_errno : 0,
                       "could not patch %s%s", filename,
                       retcode == -1 ? "" : "; will refetch");

                patch_failed = 1;
            }
            free (backup);
#endif /* Use patch. */
        }
        else
        {
            char *filebuf;
            size_t filebufsize;
            size_t nread;
            char *patchedbuf;
            size_t patchedlen;

            /* Handle UPDATE_ENTRIES_RCS_DIFF. */

            if (!isfile (filename))
                error (1, 0, "patch original file %s does not exist",
                       short_pathname);
            filebuf = NULL;
            filebufsize = 0;
            nread = 0;

            get_file (filename, short_pathname, bin ? FOPEN_BINARY_READ : "r",
                      &filebuf, &filebufsize, &nread);
            /* At this point the contents of the existing file are in
               FILEBUF, and the length of the contents is in NREAD.
               The contents of the patch from the network are in BUF,
               and the length of the patch is in SIZE. */

            if (! rcs_change_text (short_pathname, filebuf, nread, buf, size,
                                   &patchedbuf, &patchedlen))
                patch_failed = 1;
            else
            {
                if (stored_checksum_valid)
                {
                    struct MD5Context context;
                    unsigned char checksum[16];

                    /* We have a checksum.  Check it before writing
                       the file out, so that we don't have to read it
                       back in again. */
                    MD5Init (&context);
                    MD5Update (&context, (unsigned char *) patchedbuf, patchedlen);
                    MD5Final (checksum, &context);
                    if (memcmp (checksum, stored_checksum, 16) != 0)
                    {
```

```
                    error (0, 0,
1890                        "checksum failure after patch to %s; will refetch",
                            short_pathname);

                    patch_failed = 1;
                }

                stored_checksum_valid = 0;
            }

            if (! patch_failed)
1900        {
                FILE *e;

                e = open_file (temp_filename,
                               bin ? FOPEN_BINARY_WRITE : "w");
                if (fwrite (patchedbuf, 1, patchedlen, e) != patchedlen)
                    error (1, errno, "cannot write %s", temp_filename);
                if (fclose (e) == EOF)
                    error (1, errno, "cannot close %s", temp_filename);
                rename_file (temp_filename, filename);
1910        }

            free (patchedbuf);
        }

        free (filebuf);
    }

    free (temp_filename);

1920    if (stored_checksum_valid && ! patch_failed)
    {
        FILE *e;
        struct MD5Context context;
        unsigned char buf[8192];
        unsigned len;
        unsigned char checksum[16];

        /*
         * Compute the MD5 checksum.  This will normally only be
1930     * used when receiving a patch, so we always compute it
         * here on the final file, rather than on the received
         * data.
         *
         * Note that if the file is a text file, we should read it
         * here using text mode, so its lines will be terminated the same
         * way they were transmitted.
         */
        e = CVS_FOPEN (filename, "r");
        if (e == NULL)
1940        error (1, errno, "could not open %s", short_pathname);

        MD5Init (&context);
        while ((len = fread (buf, 1, sizeof buf, e)) != 0)
            MD5Update (&context, buf, len);
        if (ferror (e))
            error (1, errno, "could not read %s", short_pathname);
        MD5Final (checksum, &context);

        fclose (e);
1950
        stored_checksum_valid = 0;

        if (memcmp (checksum, stored_checksum, 16) != 0)
        {
            if (data->contents != UPDATE_ENTRIES_PATCH)
                error (1, 0, "checksum failure on %s",
                       short_pathname);

            error (0, 0,
1960               "checksum failure after patch to %s; will refetch",
                   short_pathname);

            patch_failed = 1;
        }
    }

    if (patch_failed)
    {
        /* Save this file to retrieve later.  */
1970    failed_patches = (char **) xrealloc ((char *) failed_patches,
                                             ((failed_patches_count + 1)
                                              * sizeof (char *)));
        failed_patches[failed_patches_count] = xstrdup (short_pathname);
        ++failed_patches_count;

        stored_checksum_valid = 0;

        free (mode_string);
```

```
            free (buf);

            return;
        }

        {
            int status = change_mode (filename, mode_string, 1);
            if (status != 0)
                error (0, status, "cannot change mode of %s", short_pathname);
        }

        free (mode_string);
        free (buf);
    }

    if (stored_mode_valid)
        change_mode (filename, stored_mode, 1);
    stored_mode_valid = 0;

    if (stored_modtime_valid)
    {
        struct utimbuf t;

        memset (&t, 0, sizeof (t));
        /* There is probably little point in trying to preserved the
           actime (or is there? What about Checked-in?). */
        t.modtime = t.actime = stored_modtime;

#ifdef UTIME_EXPECTS_WRITABLE
        if (!iswritable (filename))
        {
            xchmod (filename, 1);
            change_it_back = 1;
        }
#endif /* UTIME_EXPECTS_WRITABLE */

        if (utime (filename, &t) < 0)
            error (0, errno, "cannot set time on %s", filename);

#ifdef UTIME_EXPECTS_WRITABLE
        if (change_it_back == 1)
        {
            xchmod (filename, 0);
            change_it_back = 0;
        }
#endif /* UTIME_EXPECTS_WRITABLE */

        stored_modtime_valid = 0;
    }

    /*
     * Process the entries line.  Do this after we've written the file,
     * since we need the timestamp.
     */
    if (strcmp (command_name, "export") != 0)
    {
        char *local_timestamp;
        char *file_timestamp;

        (void) time (&last_register_time);

        local_timestamp = data->timestamp;
        if (local_timestamp == NULL || ts[0] == '+')
            file_timestamp = time_stamp (filename);
        else
            file_timestamp = NULL;

        /*
         * These special version numbers signify that it is not up to
         * date.  Create a dummy timestamp which will never compare
         * equal to the timestamp of the file.
         */
        if (vn[0] == '\0' || vn[0] == '0' || vn[0] == '-')
            local_timestamp = "dummy timestamp";
        else if (local_timestamp == NULL)
        {
            local_timestamp = file_timestamp;
            mark_up_to_date (filename);
        }

        /* Since we didn't get root from the server (as noted above)
           the root we write out to the entries file is what we think the root is
           at this point */
        /* Don't touch entries if we are fetching a remote rev */
        if (!client_active || !fetching_remote) {
        Register (ent_list, filename, vn, local_timestamp,
                  options, tag, date, ts[0] == '+' ? file_timestamp : NULL, CVSroot_original, repository);
        } else if (fetching_remote) {
          FILE* remote_revs_file = fopen (CVSADM_REMOTES, "a+");
          if (remote_revs_file != NULL) {
```

```
            fprintf (remote_revs_file, "%s/%s/%s\n", filename, vn, strchr (remote_file, '/') + 1);
            fclose (remote_revs_file);
          }
        }

        if (file_timestamp)
          free (file_timestamp);

        free (scratch_entries);
      }
      free (entries_line);
}

static void
handle_checked_in (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_CHECKIN;
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void
handle_new_entry (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_CHECKIN;
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = "dummy timestamp from new-entry";
    call_in_directory (args, update_entries, (char *)&dat);
}

static void
handle_updated (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_UPDATE;
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void handle_created PROTO((char *, int));

static void
handle_created (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_UPDATE;
    dat.existp = UPDATE_ENTRIES_NEW;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void handle_update_existing PROTO((char *, int));

static void
handle_update_existing (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_UPDATE;
    dat.existp = UPDATE_ENTRIES_EXISTING;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void
handle_merged (args, len)
    char *args;
    int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_UPDATE;
    /* Think this could be UPDATE_ENTRIES_EXISTING, but just in case...  */
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = "Result of merge";
    call_in_directory (args, update_entries, (char *)&dat);
}
```

```
static void
handle_patched (args, len)
     char *args;
     int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_PATCH;
    /* Think this could be UPDATE_ENTRIES_EXISTING, but just in case... */
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void
handle_rcs_diff (args, len)
     char *args;
     int len;
{
    struct update_entries_data dat;
    dat.contents = UPDATE_ENTRIES_RCS_DIFF;
    /* Think this could be UPDATE_ENTRIES_EXISTING, but just in case... */
    dat.existp = UPDATE_ENTRIES_EXISTING_OR_NEW;
    dat.timestamp = NULL;
    call_in_directory (args, update_entries, (char *)&dat);
}

static void
remove_entry (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    Scratch_Entry (ent_list, filename);
}

static void
handle_remove_entry (args, len)
    char *args;
    int len;
{
    call_in_directory (args, remove_entry, (char *)NULL);
}

static void
remove_entry_and_file (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    Scratch_Entry (ent_list, filename);
    /* Note that we don't ignore existence_error's here.  The server
       should be sending Remove-entry rather than Removed in cases
       where the file does not exist.  And if the user removes the
       file halfway through a cvs command, we should be printing an
       error.  */
    if (unlink_file (filename) < 0)
        error (0, errno, "unable to remove %s", short_pathname);
}

static void
handle_removed (args, len)
    char *args;
    int len;
{
    call_in_directory (args, remove_entry_and_file, (char *)NULL);
}

/* Is this the top level (directory containing CVSROOT)?  */
static int
is_cvsroot_level (pathname)
    char *pathname;
{
    if (strcmp (toplevel_repos, CVSroot_directory) != 0)
        return 0;

    return strchr (pathname, '/') == NULL;
}

static void
set_static (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    FILE *fp;
    fp = open_file (CVSADM_ENTSTAT, "w+");
    if (fclose (fp) == EOF)
```

```
        error (1, errno, "cannot close %s", CVSADM_ENTSTAT);
}

static void
handle_set_static_directory (args, len)
    char *args;
    int len;
{
    if (strcmp (command_name, "export") == 0)
    {
        /* Swallow the repository.  */
        read_line (NULL);
        return;
    }
    call_in_directory (args, set_static, (char *)NULL);
}

static void
clear_static (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    if (unlink_file (CVSADM_ENTSTAT) < 0 && ! existence_error (errno))
        error (1, errno, "cannot remove file %s", CVSADM_ENTSTAT);
}

static void
handle_clear_static_directory (pathname, len)
    char *pathname;
    int len;
{
    if (strcmp (command_name, "export") == 0)
    {
        /* Swallow the repository.  */
        read_line (NULL);
        return;
    }

    if (is_cvsroot_level (pathname))
    {
        /*
         * Top level (directory containing CVSROOT).  This seems to normally
         * lack a CVS directory, so don't try to create files in it.
         */
        return;
    }
    call_in_directory (pathname, clear_static, (char *)NULL);
}

static void
set_sticky (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    char *tagspec;
    FILE *f;

    read_line (&tagspec);
    f = open_file (CVSADM_TAG, "w+");
    if (fprintf (f, "%s\n", tagspec) < 0)
        error (1, errno, "writing %s", CVSADM_TAG);
    if (fclose (f) == EOF)
        error (1, errno, "closing %s", CVSADM_TAG);
    free (tagspec);
}

static void
handle_set_sticky (pathname, len)
    char *pathname;
    int len;
{
    if (strcmp (command_name, "export") == 0)
    {
        /* Swallow the repository.  */
        read_line (NULL);
        /* Swallow the tag line.  */
        read_line (NULL);
        return;
    }
    if (is_cvsroot_level (pathname))
    {
        /*
         * Top level (directory containing CVSROOT).  This seems to normally
         * lack a CVS directory, so don't try to create files in it.
         */
```

```
        /* Swallow the repository.  */
        read_line (NULL);
        /* Swallow the tag line.  */
        read_line (NULL);
        return;
    }

    call_in_directory (pathname, set_sticky, (char *)NULL);
}

static void
clear_sticky (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    if (unlink_file (CVSADM_TAG) < 0 && ! existence_error (errno))
        error (1, errno, "cannot remove %s", CVSADM_TAG);
}

static void
handle_clear_sticky (pathname, len)
    char *pathname;
    int len;
{
    if (strcmp (command_name, "export") == 0)
    {
        /* Swallow the repository.  */
        read_line (NULL);
        return;
    }

    if (is_cvsroot_level (pathname))
    {
        /*
         * Top level (directory containing CVSROOT).  This seems to normally
         * lack a CVS directory, so don't try to create files in it.
         */
        return;
    }

    call_in_directory (pathname, clear_sticky, (char *)NULL);
}


static void template PROTO ((char *, List *, char *, char *));

static void
template (data, ent_list, short_pathname, filename)
    char *data;
    List *ent_list;
    char *short_pathname;
    char *filename;
{
    /* FIXME: should be computing second argument from CVSADM_TEMPLATE
       and short_pathname.  */
    read_counted_file (CVSADM_TEMPLATE, "<CVS/Template file>");
}

static void handle_template PROTO ((char *, int));

static void
handle_template (pathname, len)
    char *pathname;
    int len;
{
    call_in_directory (pathname, template, NULL);
}


struct save_prog {
    char *name;
    char *dir;
    struct save_prog *next;
};

static struct save_prog *checkin_progs;
static struct save_prog *update_progs;

/*
 * Unlike some responses this doesn't include the repository.  So we can't
 * just call call_in_directory and have the right thing happen; we save up
 * the requests and do them at the end.
 */
static void
handle_set_checkin_prog (args, len)
    char *args;
    int len;
{
```

```
    char *prog;
    struct save_prog *p;
    read_line (&prog);
    p = (struct save_prog *) xmalloc (sizeof (struct save_prog));
    p->next = checkin_progs;
    p->dir = xstrdup (args);
    p->name = prog;
    checkin_progs = p;
}

static void
handle_set_update_prog (args, len)
    char *args;
    int len;
{
    char *prog;
    struct save_prog *p;
    read_line (&prog);
    p = (struct save_prog *) xmalloc (sizeof (struct save_prog));
    p->next = update_progs;
    p->dir = xstrdup (args);
    p->name = prog;
    update_progs = p;
}

static void do_deferred_progs PROTO((void));

static void
do_deferred_progs ()
{
    struct save_prog *p;
    struct save_prog *q;

    char *fname;
    FILE *f;

    if (toplevel_wd != NULL)
    {
        if (CVS_CHDIR (toplevel_wd) < 0)
            error (1, errno, "could not chdir to %s", toplevel_wd);
    }
    for (p = checkin_progs; p != NULL; )
    {
        fname = xmalloc (strlen (p->dir) + sizeof CVSADM_CIPROG + 10);
        sprintf (fname, "%s/%s", p->dir, CVSADM_CIPROG);
        f = open_file (fname, "w");
        if (fprintf (f, "%s\n", p->name) < 0)
            error (1, errno, "writing %s", fname);
        if (fclose (f) == EOF)
            error (1, errno, "closing %s", fname);
        free (p->name);
        free (p->dir);
        q = p->next;
        free (p);
        p = q;
        free (fname);
    }
    checkin_progs = NULL;
    for (p = update_progs; p != NULL; )
    {
        fname = xmalloc (strlen (p->dir) + sizeof CVSADM_UPROG + 10);
        sprintf (fname, "%s/%s", p->dir, CVSADM_UPROG);
        f = open_file (fname, "w");
        if (fprintf (f, "%s\n", p->name) < 0)
            error (1, errno, "writing %s", fname);
        if (fclose (f) == EOF)
            error (1, errno, "closing %s", fname);
        free (p->name);
        free (p->dir);
        q = p->next;
        free (p);
        p = q;
        free (fname);
    }
    update_progs = NULL;
}

struct save_dir {
    char *dir;
    struct save_dir *next;
};

struct save_dir *prune_candidates;

static void
add_prune_candidate (dir)
    char *dir;
{
    struct save_dir *p;
```

```
    if ((dir[0] == '.' && dir[1] == '\0')
        || (prune_candidates != NULL
            && strcmp (dir, prune_candidates->dir) == 0))
        return;
    p = (struct save_dir *) xmalloc (sizeof (struct save_dir));
    p->dir = xstrdup (dir);
    p->next = prune_candidates;
    prune_candidates = p;
}

static void process_prune_candidates PROTO((void));

static void
process_prune_candidates ()
{
    struct save_dir *p;
    struct save_dir *q;

    if (toplevel_wd != NULL)
    {
        if (CVS_CHDIR (toplevel_wd) < 0)
            error (1, errno, "could not chdir to %s", toplevel_wd);
    }
    for (p = prune_candidates; p != NULL; )
    {
        if (isemptydir (p->dir, 1))
        {
            char *b;

            if (unlink_file_dir (p->dir) < 0)
                error (0, errno, "cannot remove %s", p->dir);
            b = strrchr (p->dir, '/');
            if (b == NULL)
                Subdir_Deregister ((List *) NULL, (char *) NULL, p->dir);
            else
            {
                *b = '\0';
                Subdir_Deregister ((List *) NULL, p->dir, b + 1);
            }
        }
        free (p->dir);
        q = p->next;
        free (p);
        p = q;
    }
    prune_candidates = NULL;
}

/* Send a Repository line.  */

static char *last_repos;
static char *last_update_dir;

static void send_repository PROTO((char *, char *, char *));

static void
send_repository (dir, repos, update_dir)
    char *dir;
    char *repos;
    char *update_dir;
{
    char *adm_name;

    /* FIXME: this is probably not the best place to check; I wish I
     * knew where in here's callers to really trap this bug.  To
     * reproduce the bug, just do this:
     *
     *      mkdir junk
     *      cd junk
     *      cvs -d some_repos update foo
     *
     * Poof, CVS seg faults and dies!  It's because it's trying to
     * send a NULL string to the server but dies in send_to_server.
     * That string was supposed to be the repository, but it doesn't
     * get set because there's no CVSADM dir, and somehow it's not
     * getting set from the -d argument either... ?
     */
    if (repos == NULL)
    {
        /* Lame error.  I want a real fix but can't stay up to track
           this down right now. */
        error (1, 0, "no repository");
    }

    if (update_dir == NULL || update_dir[0] == '\0')
        update_dir = ".";

    if (last_repos != NULL
        && strcmp (repos, last_repos) == 0
        && last_update_dir != NULL
```

```
        && strcmp (update_dir, last_update_dir) == 0)
2610    /* We've already sent it.  */
        return;

    if (client_prune_dirs)
        add_prune_candidate (update_dir);

    /* 80 is large enough for any of CVSADM_*.  */
    adm_name = xmalloc (strlen (dir) + 80);

    send_to_server ("Directory ", 0);
2620    {
        /* Send the directory name.  I know that this
           sort of duplicates code elsewhere, but each
           case seems slightly different...  */
        char buf[1];
        char *p = update_dir;
        while (*p != '\0')
        {
            assert (*p != '\012');
            if (ISDIRSEP (*p))
2630        {
                buf[0] = '/';
                send_to_server (buf, 1);
            }
            else
            {
                buf[0] = *p;
                send_to_server (buf, 1);
            }
            ++p;
2640    }
    }
    send_to_server ("\012", 1);
    send_to_server (repos, 0);
    send_to_server ("\012", 1);

    if (supported_request ("Static-directory"))
    {
        adm_name[0] = '\0';
        if (dir[0] != '\0')
2650    {
            strcat (adm_name, dir);
            strcat (adm_name, "/");
        }
        strcat (adm_name, CVSADM_ENTSTAT);
        if (isreadable (adm_name))
        {
            send_to_server ("Static-directory\012", 0);
        }
    }
2660    if (supported_request ("Sticky"))
    {
        FILE *f;
        if (dir[0] == '\0')
            strcpy (adm_name, CVSADM_TAG);
        else
            sprintf (adm_name, "%s/%s", dir, CVSADM_TAG);

        f = CVS_FOPEN (adm_name, "r");
        if (f == NULL)
2670    {
            if (! existence_error (errno))
                error (1, errno, "reading %s", adm_name);
        }
        else
        {
            char line[80];
            char *nl = NULL;
            send_to_server ("Sticky ", 0);
            while (fgets (line, sizeof (line), f) != NULL)
2680        {
                send_to_server (line, 0);
                nl = strchr (line, '\n');
                if (nl != NULL)
                    break;
            }
            if (nl == NULL)
                send_to_server ("\012", 1);
            if (fclose (f) == EOF)
                error (0, errno, "closing %s", adm_name);
2690    }
    }
    if (supported_request ("Checkin-prog"))
    {
        FILE *f;
        if (dir[0] == '\0')
            strcpy (adm_name, CVSADM_CIPROG);
        else
            sprintf (adm_name, "%s/%s", dir, CVSADM_CIPROG);
```

```
2700        f = CVS_FOPEN (adm_name, "r");
            if (f == NULL)
            {
                if (! existence_error (errno))
                    error (1, errno, "reading %s", adm_name);
            }
            else
            {
                char line[80];
                char *nl = NULL;
2710
                send_to_server ("Checkin-prog ", 0);

                while (fgets (line, sizeof (line), f) != NULL)
                {
                    send_to_server (line, 0);

                    nl = strchr (line, '\n');
                    if (nl != NULL)
                        break;
2720            }
                if (nl == NULL)
                    send_to_server ("\012", 1);
                if (fclose (f) == EOF)
                    error (0, errno, "closing %s", adm_name);
            }
        }
        if (supported_request ("Update-prog"))
        {
            FILE *f;
2730        if (dir[0] == '\0')
                strcpy (adm_name, CVSADM_UPROG);
            else
                sprintf (adm_name, "%s/%s", dir, CVSADM_UPROG);

            f = CVS_FOPEN (adm_name, "r");
            if (f == NULL)
            {
                if (! existence_error (errno))
                    error (1, errno, "reading %s", adm_name);
2740        }
            else
            {
                char line[80];
                char *nl = NULL;

                send_to_server ("Update-prog ", 0);

                while (fgets (line, sizeof (line), f) != NULL)
                {
2750                send_to_server (line, 0);

                    nl = strchr (line, '\n');
                    if (nl != NULL)
                        break;
                }
                if (nl == NULL)
                    send_to_server ("\012", 1);
                if (fclose (f) == EOF)
                    error (0, errno, "closing %s", adm_name);
2760        }
        }
        free (adm_name);
        if (last_repos != NULL)
            free (last_repos);
        if (last_update_dir != NULL)
            free (last_update_dir);
        last_repos = xstrdup (repos);
        last_update_dir = xstrdup (update_dir);
    }
2770
    /* Send a Repository line and set toplevel_repos.  */

    void
    send_a_repository (dir, repository, update_dir)
        char *dir;
        char *repository;
        char *update_dir;
    {
        if (toplevel_repos == NULL && repository != NULL)
2780    {
            if (update_dir[0] == '\0'
                || (update_dir[0] == '.' && update_dir[1] == '\0'))
                toplevel_repos = xstrdup (repository);
            else
            {
                /*
                 * Get the repository from a CVS/Repository file if update_dir
                 * is absolute.  This is not correct in general, because
```

```
             * the CVS/Repository file might not be the top-level one.
2790         * This is for cases like "cvs update /foo/bar" (I'm not
             * sure it matters what toplevel_repos we get, but it does
             * matter that we don't hit the "internal error" code below).
             */
            if (update_dir[0] == '/')
                toplevel_repos = Name_Repository (update_dir, update_dir);
            else
            {
                /*
                 * Guess the repository of that directory by looking at a
2800             * subdirectory and removing as many pathname components
                 * as are in update_dir.  I think that will always (or at
                 * least almost always) be 1.
                 *
                 * So this deals with directories which have been
                 * renamed, though it doesn't necessarily deal with
                 * directories which have been put inside other
                 * directories (and cvs invoked on the containing
                 * directory).  I'm not sure the latter case needs to
                 * work.
2810             */
                /*
                 * This gets toplevel_repos wrong for "cvs update ../foo"
                 * but I'm not sure toplevel_repos matters in that case.
                 */
                int slashes_in_update_dir;
                int slashes_skipped;
                char *p;

                /*
2820             * Strip trailing slashes from the name of the update directory.
                 * Otherwise, running 'cvs update dir/' provokes the failure
                 * 'protocol error: illegal directory syntax in dir/' when
                 * running in client/server mode.
                 */
                strip_trailing_slashes (update_dir);

                slashes_in_update_dir = 0;
                for (p = update_dir; *p != '\0'; ++p)
                    if (*p == '/')
2830                    ++slashes_in_update_dir;

                slashes_skipped = 0;
                p = repository + strlen (repository);
                while (1)
                {
                    if (p == repository)
                        error (1, 0,
                               "internal error: not enough slashes in %s",
                               repository);
2840                if (*p == '/')
                        ++slashes_skipped;
                    if (slashes_skipped < slashes_in_update_dir + 1)
                        --p;
                    else
                        break;
                }
                toplevel_repos = xmalloc (p - repository + 1);
                /* Note that we don't copy the trailing '/'.  */
                strncpy (toplevel_repos, repository, p - repository);
2850            toplevel_repos[p - repository] = '\0';
            }
        }
    }

    send_repository (dir, repository, update_dir);
}

/* The "expanded" modules.  */
static int modules_count;
2860 static int modules_allocated;
static char **modules_vector;

static void
handle_module_expansion (args, len)
    char *args;
    int len;
{
    if (modules_vector == NULL)
    {
2870      modules_allocated = 1; /* Small for testing */
        modules_vector = (char **) xmalloc
          (modules_allocated * sizeof (modules_vector[0]));
    }
    else if (modules_count >= modules_allocated)
    {
        modules_allocated *= 2;
        modules_vector = (char **) xrealloc
          ((char *) modules_vector,
```

```
            modules_allocated * sizeof (modules_vector[0]));
    }
    modules_vector[modules_count] = xmalloc (strlen (args) + 1);
    strcpy (modules_vector[modules_count], args);
    ++modules_count;
}

/* Original, not "expanded" modules.  */
static int module_argc;
static char **module_argv;

void
client_expand_modules (argc, argv, local)
    int argc;
    char **argv;
    int local;
{
    int errs;
    int i;

    module_argc = argc;
    module_argv = (char **) xmalloc ((argc + 1) * sizeof (module_argv[0]));
    for (i = 0; i < argc; ++i)
        module_argv[i] = xstrdup (argv[i]);
    module_argv[argc] = NULL;

    for (i = 0; i < argc; ++i)
        send_arg (argv[i]);
    send_a_repository ("", CVSroot_directory, "");

    send_to_server ("expand-modules\012", 0);

    errs = get_server_responses ();
    if (last_repos != NULL)
        free (last_repos);
    last_repos = NULL;
    if (last_update_dir != NULL)
        free (last_update_dir);
    last_update_dir = NULL;
    if (errs)
        error (errs, 0, "cannot expand modules");
}

void
client_send_expansions (local, where, build_dirs)
    int local;
    char *where;
    int build_dirs;
{
    int i;
    char *argv[1];

    /* Send the original module names.  The "expanded" module name might
       not be suitable as an argument to a co request (e.g. it might be
       the result of a -d argument in the modules file).  It might be
       cleaner if we genuinely expanded module names, all the way to a
       local directory and repository, but that isn't the way it works
       now.  */
    send_file_names (module_argc, module_argv, 0);

    for (i = 0; i < modules_count; ++i)
    {
        argv[0] = where ? where : modules_vector[i];
        if (isfile (argv[0]))
            send_files (1, argv, local, 0, build_dirs ? SEND_BUILD_DIRS : 0);
    }
    send_a_repository ("", CVSroot_directory, "");
}

void
client_nonexpanded_setup ()
{
    send_a_repository ("", CVSroot_directory, "");
}

/* Receive a cvswrappers line from the server; it must be a line
   containing an RCS option (e.g., "*.exe   -k 'b'").

   Note that this doesn't try to handle -t/-f options (which are a
   whole separate issue which noone has thought much about, as far
   as I know).

   We need to know the keyword expansion mode so we know whether to
   read the file in text or binary mode.  */

static void
handle_wrapper_rcs_option (args, len)
    char *args;
    int len;
{
```

```
    char *p;

    /* Enforce the notes in cvsclient.texi about how the response is not
       as free-form as it looks.  */
    p = strchr (args, ' ');
    if (p == NULL)
        goto handle_error;
    if (*++p != '-'
        || *++p != 'k'
        || *++p != ' '
        || *++p != '\'')
        goto handle_error;
    if (strchr (p, '\'') == NULL)
        goto handle_error;

    /* Add server-side cvswrappers line to our wrapper list. */
    wrap_add (args, 0);
    return;
 handle_error:
    error (0, errno, "protocol error: ignoring invalid wrappers %s", args);
}

static void
handle_not_carried (args, len)
    char *args;
    int len;
{
  char* server;
  char* root;
  char* repository;
  char* file = xmalloc (len + 1);

  strcpy (file, args);

  read_line (&server);
  read_line (&root);
  read_line (&repository);

  add_remote (file, server, root, repository);

  free (server);
  free (root);
  free (file);
  free (repository);
}

static void
handle_create_remote_branch (args, len)
    char *args;
    int len;
{
  char* server;
  char* root;
  char* repository;
  char* revision;
  char* file = xmalloc (len + 1);


  read_line (&server);
  read_line (&root);
  read_line (&repository);
  read_line (&revision);

  strcpy (file, args);

  add_remote_tag (file, server, root, repository, revision);
}

static void
handle_m (args, len)
    char *args;
    int len;
{
    /* In the case where stdout and stderr point to the same place,
       fflushing stderr will make output happen in the correct order.
       Often stderr will be line-buffered and this won't be needed,
       but not always (is that true?  I think the comment is probably
       based on being confused between default buffering between
       stdout and stderr.  But I'm not sure).  */
    fflush (stderr);
    fwrite (args, len, sizeof (*args), stdout);
    putc ('\n', stdout);
}

static void handle_mbinary PROTO ((char *, int));

static void
handle_mbinary (args, len)
    char *args;
```

```
    int len;
{
    char *size_string;
    size_t size;
    size_t totalread;
    size_t nread;
    size_t toread;
    char buf[8192];

    /* See comment at handle_m about (non)flush of stderr.  */

    /* Get the size.  */
    read_line (&size_string);
    size = atoi (size_string);
    free (size_string);

    /* OK, now get all the data.  The algorithm here is that we read
       as much as the network wants to give us in
       try_read_from_server, and then we output it all, and then
       repeat, until we get all the data.  */
    totalread = 0;
    while (totalread < size)
    {
        toread = size - totalread;
        if (toread > sizeof buf)
            toread = sizeof buf;

        nread = try_read_from_server (buf, toread);
        cvs_output_binary (buf, nread);
        totalread += nread;
    }
}

static void
handle_e (args, len)
    char *args;
    int len;
{
    /* In the case where stdout and stderr point to the same place,
       fflushing stdout will make output happen in the correct order.  */
    fflush (stdout);
    fwrite (args, len, sizeof (*args), stderr);
    putc ('\n', stderr);
}

/*ARGSUSED*/
static void
handle_f (args, len)
    char *args;
    int len;
{
    fflush (stderr);
}

static void handle_mt PROTO ((char *, int));

static void
handle_mt (args, len)
    char *args;
    int len;
{
    char *p;
    char *tag = args;
    char *text;

    /* See comment at handle_m for more details.  */
    fflush (stderr);

    p = strchr (args, ' ');
    if (p == NULL)
        text = NULL;
    else
    {
        *p++ = '\0';
        text = p;
    }

    switch (tag[0])
    {
        case '+':
            if (strcmp (tag, "+updated") == 0)
                updated_seen = 1;
            break;
        case '-':
            if (strcmp (tag, "-updated") == 0)
                updated_seen = 0;
            break;
        default:
            if (updated_seen)
            {
```

```
            if (strcmp (tag, "fname") == 0)
3150        {
                if (updated_fname != NULL)
                {
                    /* Output the previous message now.  This can happen
                       if there was no Update-existing or other such
                       response, due to the -n global option.  */
                    cvs_output ("U ", 0);
                    cvs_output (updated_fname, 0);
                    cvs_output ("\n", 1);
                    free (updated_fname);
3160            }
                updated_fname = xstrdup (text);
            }
            /* Swallow all other tags.  Either they are extraneous
               or they reflect future extensions that we can
               safely ignore.  */
        }
        else if (strcmp (tag, "newline") == 0)
            printf ("\n");
        else if (text != NULL)
3170        printf ("%s", text);
    }
}

#endif /* CLIENT_SUPPORT */
#if defined(CLIENT_SUPPORT) || defined(SERVER_SUPPORT)

/* This table must be writeable if the server code is included.  */
struct response responses[] =
{
3180 #ifdef CLIENT_SUPPORT
#define RSP_LINE(n, f, t, s) {n, f, t, s}
#else /* ! CLIENT_SUPPORT */
#define RSP_LINE(n, f, t, s) {n, s}
#endif /* CLIENT_SUPPORT */

    RSP_LINE("ok", handle_ok, response_type_ok, rs_essential),
    RSP_LINE("error", handle_error, response_type_error, rs_essential),
    RSP_LINE("Valid-requests", handle_valid_requests, response_type_normal,
      rs_essential),
3190  RSP_LINE("Checked-in", handle_checked_in, response_type_normal,
      rs_essential),
    RSP_LINE("New-entry", handle_new_entry, response_type_normal, rs_optional),
    RSP_LINE("Checksum", handle_checksum, response_type_normal, rs_optional),
    RSP_LINE("Copy-file", handle_copy_file, response_type_normal, rs_optional),
    RSP_LINE("Updated", handle_updated, response_type_normal, rs_essential),
    RSP_LINE("Created", handle_created, response_type_normal, rs_optional),
    RSP_LINE("Update-existing", handle_update_existing, response_type_normal,
      rs_optional),
    RSP_LINE("Merged", handle_merged, response_type_normal, rs_essential),
3200  RSP_LINE("Patched", handle_patched, response_type_normal, rs_optional),
    RSP_LINE("Rcs-diff", handle_rcs_diff, response_type_normal, rs_optional),
    RSP_LINE("Mode", handle_mode, response_type_normal, rs_optional),
    RSP_LINE("Mod-time", handle_mod_time, response_type_normal, rs_optional),
    RSP_LINE("Removed", handle_removed, response_type_normal, rs_essential),
    RSP_LINE("Remove-entry", handle_remove_entry, response_type_normal,
      rs_optional),
    RSP_LINE("Set-static-directory", handle_set_static_directory,
      response_type_normal,
      rs_optional),
3210  RSP_LINE("Clear-static-directory", handle_clear_static_directory,
      response_type_normal,
      rs_optional),
    RSP_LINE("Set-sticky", handle_set_sticky, response_type_normal,
      rs_optional),
    RSP_LINE("Clear-sticky", handle_clear_sticky, response_type_normal,
      rs_optional),
    RSP_LINE("Template", handle_template, response_type_normal,
      rs_optional),
    RSP_LINE("Set-checkin-prog", handle_set_checkin_prog, response_type_normal,
3220    rs_optional),
    RSP_LINE("Set-update-prog", handle_set_update_prog, response_type_normal,
      rs_optional),
    RSP_LINE("Notified", handle_notified, response_type_normal, rs_optional),
    RSP_LINE("Module-expansion", handle_module_expansion, response_type_normal,
      rs_optional),
    RSP_LINE("Wrapper-rcsOption", handle_wrapper_rcs_option,
      response_type_normal,
      rs_optional),
    RSP_LINE("Not-carried", handle_not_carried, response_type_normal, rs_essential),
3230  RSP_LINE("Create-remote-branch", handle_create_remote_branch, response_type_normal, rs_essential),
    RSP_LINE("M", handle_m, response_type_normal, rs_essential),
    RSP_LINE("Mbinary", handle_mbinary, response_type_normal, rs_optional),
    RSP_LINE("E", handle_e, response_type_normal, rs_essential),
    RSP_LINE("F", handle_f, response_type_normal, rs_optional),
    RSP_LINE("MT", handle_mt, response_type_normal, rs_optional),
    /* Possibly should be response_type_error.  */
    RSP_LINE(NULL, NULL, response_type_normal, rs_essential)
```

```
#undef RSP_LINE
};

#endif /* CLIENT_SUPPORT or SERVER_SUPPORT */
#ifdef CLIENT_SUPPORT

/*
 * If LEN is 0, then send_to_server() computes string's length itself.
 *
 * Therefore, pass the real length when transmitting data that might
 * contain 0's.
 */
void
send_to_server (str, len)
     char *str;
     size_t len;
{
    static int nbytes;

    if (len == 0)
        len = strlen (str);

    buf_output (to_server, str, len);

    /* There is no reason not to send data to the server, so do it
       whenever we've accumulated enough information in the buffer to
       make it worth sending.  */
    nbytes += len;
    if (nbytes >= 2 * BUFFER_DATA_SIZE)
    {
        int status;

        status = buf_send_output (to_server);
        if (status != 0)
            error (1, status, "error writing to server");
        nbytes = 0;
    }
}

/* Read up to LEN bytes from the server.  Returns actual number of
   bytes read, which will always be at least one; blocks if there is
   no data available at all.  Gives a fatal error on EOF or error.  */
static size_t
try_read_from_server (buf, len)
    char *buf;
    size_t len;
{
    int status, nread;
    char *data;

    status = buf_read_data (from_server, len, &data, &nread);
    if (status != 0)
    {
        if (status == -1)
            error (1, 0,
                   "end of file from server (consult above messages if any)");
        else if (status == -2)
            error (1, 0, "out of memory");
        else
            error (1, status, "reading from server");
    }

    memcpy (buf, data, nread);

    return nread;
}

/*
 * Read LEN bytes from the server or die trying.
 */
void
read_from_server (buf, len)
    char *buf;
    size_t len;
{
    size_t red = 0;
    while (red < len)
    {
        red += try_read_from_server (buf + red, len - red);
        if (red == len)
            break;
    }
}

/*
 * Get some server responses and process them.  Returns nonzero for
 * error, 0 for success.  */
int
get_server_responses ()
{
```

```
    struct response *rs;
    do
    {
        char *cmd;
        int len;

        len = read_line (&cmd);
        for (rs = responses; rs->name != NULL; ++rs)
            if (strncmp (cmd, rs->name, strlen (rs->name)) == 0)
            {
                int cmdlen = strlen (rs->name);
                if (cmd[cmdlen] == '\0')
                    ;
                else if (cmd[cmdlen] == ' ')
                    ++cmdlen;
                else
                    /*
                     * The first len characters match, but it's a different
                     * response.  e.g. the response is "oklahoma" but we
                     * matched "ok".
                     */
                    continue;
                (*rs->func) (cmd + cmdlen, len - cmdlen);
                break;
            }
        if (rs->name == NULL)
            /* It's OK to print just to the first '\0'.  */
            /* We might want to handle control characters and the like
               in some other way other than just sending them to stdout.
               One common reason for this error is if people use :ext:
               with a version of rsh which is doing CRLF translation or
               something, and so the client gets "ok^M" instead of "ok".
               Right now that will tend to print part of this error
               message over the other part of it.  It seems like we could
               do better (either in general, by quoting or omitting all
               control characters, and/or specifically, by detecting the CRLF
               case and printing a specific error message).  */
            error (0, 0,
                   "warning: unrecognized response `%s' from cvs server",
                   cmd);
        free (cmd);
    } while (rs->type == response_type_normal);

    if (updated_fname != NULL)
    {
        /* Output the previous message now.  This can happen
           if there was no Update-existing or other such
           response, due to the -n global option.  */
        cvs_output ("U ", 0);
        cvs_output (updated_fname, 0);
        cvs_output ("\n", 1);
        free (updated_fname);
        updated_fname = NULL;
    }

    if (rs->type == response_type_error)
        return 1;
    if (failure_exit)
        return 1;
    return 0;
}

/* Get the responses and then close the connection.  */
int server_fd = -1;

/*
 * Flag var; we'll set it in start_server() and not one of its
 * callees, such as start_rsh_server().  This means that there might
 * be a small window between the starting of the server and the
 * setting of this var, but all the code in that window shouldn't care
 * because it's busy checking return values to see if the server got
 * started successfully anyway.
 */
int server_started = 0;

int
get_responses_and_close ()
{
    int errs = get_server_responses ();
    int status;

    if (last_entries != NULL)
    {
        Entries_Close (last_entries);
        last_entries = NULL;
    }

    do_deferred_progs ();

    if (client_prune_dirs)
```

```
        process_prune_candidates ();

    /* The calls to buf_shutdown are currently only meaningful when we
       are using compression.  First we shut down TO_SERVER.  That
       tells the server that its input is finished.  It then shuts
       down the buffer it is sending to us, at which point our shut
       down of FROM_SERVER will complete.  */

    status = buf_shutdown (to_server);
    if (status != 0)
        error (0, status, "shutting down buffer to server");
    status = buf_shutdown (from_server);
    if (status != 0)
        error (0, status, "shutting down buffer from server");

#ifdef NO_SOCKET_TO_FD
    if (use_socket_style)
    {
        if (shutdown (server_sock, 2) < 0)
            error (1, 0, "shutting down server socket: %s", SOCK_STRERROR (SOCK_ERRNO));
    }
    else
#endif /* NO_SOCKET_TO_FD */
    {
#if defined(HAVE_KERBEROS) || defined(AUTH_CLIENT_SUPPORT)
        if (server_fd != -1)
        {
            if (shutdown (server_fd, 1) < 0)
                error (1, 0, "shutting down connection to %s: %s",
                       CVSroot_hostname, SOCK_STRERROR (SOCK_ERRNO));
            /*
             * This test will always be true because we dup the descriptor
             */
            if (fileno (from_server_fp) != fileno (to_server_fp))
            {
                if (fclose (to_server_fp) != 0)
                    error (1, errno,
                           "closing down connection to %s",
                           CVSroot_hostname);
            }
        }
        else
#endif

#ifdef SHUTDOWN_SERVER
            SHUTDOWN_SERVER (fileno (to_server_fp));
#else /* ! SHUTDOWN_SERVER */
        {

#ifdef START_RSH_WITH_POPEN_RW
            if (pclose (to_server_fp) == EOF)
#else /* ! START_RSH_WITH_POPEN_RW */
                if (fclose (to_server_fp) == EOF)
#endif /* START_RSH_WITH_POPEN_RW */
                {
                    error (1, errno, "closing connection to %s",
                           CVSroot_hostname);
                }
        }

        if (! buf_empty_p (from_server)
            || getc (from_server_fp) != EOF)
            error (0, 0, "dying gasps from %s unexpected", CVSroot_hostname);
        else if (ferror (from_server_fp))
            error (0, errno, "reading from %s", CVSroot_hostname);

        fclose (from_server_fp);
#endif /* SHUTDOWN_SERVER */
    }

    if (rsh_pid != -1
        && waitpid (rsh_pid, (int *) 0, 0) == -1)
        error (1, errno, "waiting for process %d", rsh_pid);

    server_started = 0;

    /* see if we need to sleep before returning */
    if (last_register_time)
    {
        time_t now;

        (void) time (&now);
        if (now == last_register_time)
            sleep (1);                  /* to avoid time-stamp races */
    }

    return errs;
}

#ifndef NO_EXT_METHOD
```

```
static void start_rsh_server PROTO((int *, int *));
#endif

int
supported_request (name)
    char *name;
{
    struct request *rq;

    /* Special - do not use update-patches when fetchinf deferred remotes */
    if (strcmp (name, "update-patches") == 0)
      return 0;

    for (rq = requests; rq->name; rq++)
        if (!strcmp (rq->name, name))
            return rq->status == rq_supported;
    error (1, 0, "internal error: testing support for unknown option?");
    /* NOTREACHED */
    return 0;
}

#if defined (AUTH_CLIENT_SUPPORT) || defined (HAVE_KERBEROS)
static struct hostent *init_sockaddr PROTO ((struct sockaddr_in *, char *,
                                             unsigned int));

static struct hostent *
init_sockaddr (name, hostname, port)
    struct sockaddr_in *name;
    char *hostname;
    unsigned int port;
{
    struct hostent *hostinfo;
    unsigned short shortport = port;

    memset (name, 0, sizeof (*name));
    name->sin_family = AF_INET;
    name->sin_port = htons (shortport);
    hostinfo = gethostbyname (hostname);
    if (hostinfo == NULL)
    {
        fprintf (stderr, "Unknown host %s.\n", hostname);
        error_exit ();
    }
    name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
    return hostinfo;
}

#endif /* defined (AUTH_CLIENT_SUPPORT) || defined (HAVE_KERBEROS) */

#ifdef AUTH_CLIENT_SUPPORT

static int auth_server_port_number PROTO ((void));

static int
auth_server_port_number ()
{
    struct servent *s = getservbyname ("cvspserver", "tcp");

    if (s)
        return ntohs (s->s_port);
    else
        return CVS_AUTH_PORT;
}


/* Read a line from socket SOCK.  Result does not include the
   terminating linefeed.  This is only used by the authentication
   protocol, which we call before we set up all the buffering stuff.
   It is possible it should use the buffers too, which would be faster
   (unlike the server, there isn't really a security issue in terms of
   separating authentication from the rest of the code).

   Space for the result is malloc'd and should be freed by the caller.

   Returns number of bytes read.  */
static int
recv_line (sock, resultp)
    int sock;
    char **resultp;
{
    int c;
    char *result;
    size_t input_index = 0;
    size_t result_size = 80;

    result = (char *) xmalloc (result_size);

    while (1)
    {
```

```
        char ch;
        if (recv (sock, &ch, 1, 0) < 0)
            error (1, 0, "recv() from server %s: %s", CVSroot_hostname,
                   SOCK_STRERROR (SOCK_ERRNO));
        c = ch;

        if (c == EOF)
        {
            free (result);

            /* It's end of file.  */
            error (1, 0, "end of file from server");
        }

        if (c == '\012')
            break;

        result[input_index++] = c;
        while (input_index + 1 >= result_size)
        {
            result_size *= 2;
            result = (char *) xrealloc (result, result_size);
        }
    }

    if (resultp)
        *resultp = result;

    /* Terminate it just for kicks, but we *can* deal with embedded NULs.  */
    result[input_index] = '\0';

    if (resultp == NULL)
        free (result);
    return input_index;
}

/* Connect to the authenticating server.

   If VERIFY_ONLY is non-zero, then just verify that the password is
   correct and then shutdown the connection.

   If VERIFY_ONLY is 0, then really connect to the server.

   If DO_GSSAPI is non-zero, then we use GSSAPI authentication rather
   than the pserver password authentication.

   If we fail to connect or if access is denied, then die with fatal
   error.  */
void
connect_to_server (tofdp, fromfdp, verify_only, method)
     int *tofdp, *fromfdp;
     int verify_only;
     int method;
{
    int sock;
#ifndef NO_SOCKET_TO_FD
    int tofd, fromfd;
#endif
    int port_number;
    struct sockaddr_in client_sai;
    struct hostent *hostinfo;
    char* auth_list_buf;
    char* method_name;
    char* request;
    char start [64];
    char end [64];

    sock = socket (AF_INET, SOCK_STREAM, 0);
    if (sock == -1)
    {
        error (1, 0, "cannot create socket: %s", SOCK_STRERROR (SOCK_ERRNO));
    }
    port_number = auth_server_port_number ();
    hostinfo = init_sockaddr (&client_sai, CVSroot_hostname, port_number);
    if (connect (sock, (struct sockaddr *) &client_sai, sizeof (client_sai))
        < 0)
        error (1, 0, "connect to %s:%d failed: %s", CVSroot_hostname,
               port_number, SOCK_STRERROR (SOCK_ERRNO));

    recv_line (sock, &auth_list_buf);
    free (auth_list_buf);

    if (method == AUTH_GSSAPI) {
      method_name = "GSSAPI";
    } else if (method == AUTH_KERBEROS_V4) {
      method_name = "KERBEROS_V4";
    } else {
      method_name = "PASSWORD";
        }
```

```
    if (verify_only) {
3690  request = "VERIFICATION";
    } else {
      request = "AUTHENTICATION";
    }

    sprintf (start, "BEGIN %s %s REQUEST\012", method_name, request);
    sprintf (end, "END %s REQUEST\012", request);

    if (send (sock, start, strlen (start), 0) < 0) {
      error (1, 0, "cannot send: %s, ", SOCK_STRERROR (SOCK_ERRNO));
3700  }

    if (method == AUTH_GSSAPI) {
#ifdef HAVE_GSSAPI
      if (!connect_to_gserver (verify_only, sock, hostinfo))
        goto rejected;
#else
      error (1, 0, "This client does not support GSSAPI authentication");
#endif
    } else if (method == AUTH_KERBEROS_V4) {
3710 #ifdef HAVE_KERBEROS
      if (!connect_to_kserver (verify_only, sock, hostinfo))
        goto rejected;
#else
      error (1, 0, "This client does not support Kerberos v4 authentication");
#endif
    } else {
      if (!connect_to_pserver (verify_only, sock, hostinfo))
        goto rejected;
    }
3720
    if (send (sock, end, strlen (end), 0) < 0) {
      error (1, 0, "cannot send: %s, ", SOCK_STRERROR (SOCK_ERRNO));
    }

    {
        char *read_buf;

        /* Loop, getting responses from the server.  */
        while (1)
3730    {
            recv_line (sock, &read_buf);

            if (strcmp (read_buf, "I HATE YOU") == 0)
            {
                /* Authorization not granted. */
                goto rejected;
            }
            else if (strncmp (read_buf, "E ", 2) == 0)
            {
3740            fprintf (stderr, "%s\n", read_buf + 2);

                /* Continue with the authentication protocol.  */
            }
            else if (strncmp (read_buf, "error ", 6) == 0)
            {
                char *p;

                /* First skip the code.  */
                p = read_buf + 6;
3750            while (*p != ' ' && *p != '\0')
                    ++p;

                /* Skip the space that follows the code.  */
                if (*p == ' ')
                    ++p;

                /* Now output the text.  */
                fprintf (stderr, "%s\n", p);
                goto rejected;
3760        }
            else if (strcmp (read_buf, "I LOVE YOU") == 0)
            {
                free (read_buf);
                break;
            }
            else
            {
                /* Unrecognized response from server. */
                if (shutdown (sock, 2) < 0)
3770            {
                    error (0, 0,
                           "unrecognized auth response from %s: %s",
                           CVSroot_hostname, read_buf);
                    error (1, 0,
                           "shutdown() failed, server %s: %s",
                           CVSroot_hostname,
                           SOCK_STRERROR (SOCK_ERRNO));
                }
```

```
                error (1, 0,
3780                    "unrecognized auth response from %s: %s",
                        CVSroot_hostname, read_buf);
                }
                free (read_buf);
            }
        }

        if (verify_only)
        {
            if (shutdown (sock, 2) < 0)
3790            error (0, 0, "shutdown() failed, server %s: %s", CVSroot_hostname,
                       SOCK_STRERROR (SOCK_ERRNO));
            return;
        }
        else
        {
    #ifdef NO_SOCKET_TO_FD
            use_socket_style = 1;
            server_sock = sock;
            /* Try to break mistaken callers: */
3800        *tofdp = 0;
            *fromfdp = 0;
    #else /* ! NO_SOCKET_TO_FD */
            server_fd = sock;
            close_on_exec (server_fd);
            tofd = fromfd = sock;
            /* Hand them back to the caller. */
            *tofdp   = tofd;
            *fromfdp = fromfd;
    #endif /* NO_SOCKET_TO_FD */
3810    }

        return;

      rejected:
        if (shutdown (sock, 2) < 0)
        {
            error (0, 0,
                   "authorization failed: server %s rejected access",
                   CVSroot_hostname);
3820        error (1, 0,
                   "shutdown() failed (server %s): %s",
                   CVSroot_hostname,
                   SOCK_STRERROR (SOCK_ERRNO));
        }

        error (1, 0,
               "authorization failed: server %s rejected access",
               CVSroot_hostname);
    }
3830 #endif /* AUTH_CLIENT_SUPPORT */


    #if HAVE_KERBEROS

    /* This function has not been changed to deal with NO_SOCKET_TO_FD
       (i.e., systems on which sockets cannot be converted to file
       descriptors).  The first person to try building a kerberos client
       on such a system (OS/2, Windows 95, and maybe others) will have to
       make take care of this.  */
3840 void
    start_tcp_server (tofdp, fromfdp)
        int *tofdp, *fromfdp;
    {
        int s;
        const char *portenv;
        int port;
        struct hostent *hp;
        struct sockaddr_in sin;
        char *hname;
3850
        s = socket (AF_INET, SOCK_STREAM, 0);
        if (s < 0)
            error (1, 0, "cannot create socket: %s", SOCK_STRERROR (SOCK_ERRNO));

        /* Get CVS_CLIENT_PORT or look up cvs/tcp with CVS_PORT as default */
        portenv = getenv ("CVS_CLIENT_PORT");
        if (portenv != NULL)
        {
            port = atoi (portenv);
3860        if (port <= 0)
            {
                error (0, 0, "CVS_CLIENT_PORT must be a positive number!  If you");
                error (0, 0, "are trying to force a connection via rsh, please");
                error (0, 0, "put \":server:\" at the beginning of your CVSROOT");
                error (1, 0, "variable.");
            }
            if (trace)
                fprintf(stderr, "Using TCP port %d to contact server.\n", port);
```

```
    }
3870    else
    {
        struct servent *sp;

        sp = getservbyname ("cvs", "tcp");
        if (sp == NULL)
            port = CVS_PORT;
        else
            port = ntohs (sp->s_port);
    }
3880
    hp = init_sockaddr (&sin, CVSroot_hostname, port);

    hname = xmalloc (strlen (hp->h_name) + 1);
    strcpy (hname, hp->h_name);

    if (connect (s, (struct sockaddr *) &sin, sizeof sin) < 0)
        error (1, 0, "connect to %s:%d failed: %s", CVSroot_hostname,
               port, SOCK_STRERROR (SOCK_ERRNO));

3890 #ifdef HAVE_KERBEROS
    {
        const char *realm;
        struct sockaddr_in laddr;
        int laddrlen;
        KTEXT_ST ticket;
        MSG_DAT msg_data;
        CREDENTIALS cred;
        int status;

3900    realm = krb_realmofhost (hname);

        laddrlen = sizeof (laddr);
        if (getsockname (s, (struct sockaddr *) &laddr, &laddrlen) < 0)
            error (1, 0, "getsockname failed: %s", SOCK_STRERROR (SOCK_ERRNO));

        /* We don't care about the checksum, and pass it as zero.  */
        status = krb_sendauth (KOPT_DO_MUTUAL, s, &ticket, "rcmd",
                               hname, realm, (unsigned long) 0, &msg_data,
                               &cred, sched, &laddr, &sin, "KCVSV1.0");
3910    if (status != KSUCCESS)
            error (1, 0, "kerberos authentication failed: %s",
                   krb_get_err_text (status));
        memcpy (kblock, cred.session, sizeof (C_Block));
    }
#endif /* HAVE_KERBEROS */

    server_fd = s;
    close_on_exec (server_fd);

3920    free (hname);

    /* Give caller the values it wants.  */
    *tofdp   = s;
    *fromfdp = s;
}

#endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI
3930
/* Receive a given number of bytes.  */

static void
recv_bytes (sock, buf, need)
     int sock;
     char *buf;
     int need;
{
    while (need > 0)
3940    {
        int got;

        got = recv (sock, buf, need, 0);
        if (got < 0)
            error (1, 0, "recv() from server %s: %s", CVSroot_hostname,
                   SOCK_STRERROR (SOCK_ERRNO));
        buf += got;
        need -= got;
    }
3950 }

/* Connect to the server using GSSAPI authentication.  */

static int
connect_to_gserver (only_verify, sock, hostinfo)
        int only_verify; /* XXX not implemented */
     int sock;
     struct hostent *hostinfo;
```

```
{
    char *str;
    char buf[1024];
    gss_buffer_desc *tok_in_ptr, tok_in, tok_out;
    OM_uint32 stat_min, stat_maj;
    gss_name_t server_name;

    str = "BEGIN GSSAPI AUTHENTICATION REQUEST\012";

    if (send (sock, str, strlen (str), 0) < 0)
        error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));

    sprintf (buf, "cvs@%s", hostinfo->h_name);
    tok_in.length = strlen (buf);
    tok_in.value = buf;
    gss_import_name (&stat_min, &tok_in, GSS_C_NT_HOSTBASED_SERVICE,
                     &server_name);

    tok_in_ptr = GSS_C_NO_BUFFER;
    gcontext = GSS_C_NO_CONTEXT;

    do
    {
        stat_maj = gss_init_sec_context (&stat_min, GSS_C_NO_CREDENTIAL,
                                         &gcontext, server_name,
                                         GSS_C_NULL_OID,
                                         (GSS_C_MUTUAL_FLAG
                                          | GSS_C_REPLAY_FLAG),
                                         0, NULL, tok_in_ptr, NULL, &tok_out,
                                         NULL, NULL);
        if (stat_maj != GSS_S_COMPLETE && stat_maj != GSS_S_CONTINUE_NEEDED)
        {
            OM_uint32 message_context;

            message_context = 0;
            gss_display_status (&stat_min, stat_maj, GSS_C_GSS_CODE,
                                GSS_C_NULL_OID, &message_context, &tok_out);
            error (1, 0, "GSSAPI authentication failed: %s",
                   (char *) tok_out.value);
        }

        if (tok_out.length == 0)
        {
            tok_in.length = 0;
        }
        else
        {
            char cbuf[2];
            int need;

            cbuf[0] = (tok_out.length >> 8) & 0xff;
            cbuf[1] = tok_out.length & 0xff;
            if (send (sock, cbuf, 2, 0) < 0)
                error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
            if (send (sock, tok_out.value, tok_out.length, 0) < 0)
                error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));

            recv_bytes (sock, cbuf, 2);
            need = ((cbuf[0] & 0xff) << 8) | (cbuf[1] & 0xff);
            assert (need <= sizeof buf);
            recv_bytes (sock, buf, need);
            tok_in.length = need;
        }

        tok_in.value = buf;
        tok_in_ptr = &tok_in;
    }
    while (stat_maj == GSS_S_CONTINUE_NEEDED);

    return 1; /* success */
}

#endif /* HAVE_GSSAPI */

int connect_to_pserver (int verify_only, int sock, struct hostent* hostinfo)
{
        char *repository = CVSroot_directory;
        char *username   = CVSroot_username;
        char *password   = NULL;

        /* Get the password, probably from ~/.cvspass. */
        password = get_cvs_password ();

        /* Send the data the server needs. */
        if (send (sock, repository, strlen (repository), 0) < 0)
            error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
        if (send (sock, "\012", 1, 0) < 0)
            error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
        if (send (sock, username, strlen (username), 0) < 0)
            error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
```

```
        if (send (sock, "\012", 1, 0) < 0)
4050        error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
        if (send (sock, password, strlen (password), 0) < 0)
            error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));
        if (send (sock, "\012", 1, 0) < 0)
            error (1, 0, "cannot send: %s", SOCK_STRERROR (SOCK_ERRNO));

        /* Paranoia. */
        memset (password, 0, strlen (password));
        return 1; /* success */
    }
4060
    #ifdef HAVE_KERBEROS
    int connect_to_kserver (int verify_only, int sock, struct hostent* hostinfo)
    {
        char *realm;
        struct sockaddr_in laddr;
        struct sockaddr_in haddr;
        int laddrlen;
        int haddrlen;
        KTEXT_ST ticket;
4070    MSG_DAT msg_data;
        CREDENTIALS cred;
        int status;

        char *hname;

        return 1;

        hname = strdup (hostinfo -> h_name);

4080    realm = krb_realmofhost (hname);

        laddrlen = sizeof (laddr);
        if (getsockname (sock, (struct sockaddr *) &laddr, &laddrlen) < 0)
            error (1, 0, "getsockname failed: %s", SOCK_STRERROR (SOCK_ERRNO));

        haddrlen = sizeof (haddr);
        if (getpeername (sock, (struct sockaddr *) &haddr, &haddrlen) < 0)
            error (1, 0, "getpeername failed: %s", SOCK_STRERROR (SOCK_ERRNO));

4090    /* We don't care about the checksum, and pass it as zero.  */
        status = krb_sendauth (KOPT_DO_MUTUAL, sock, &ticket, "rcmd",
                               hname, realm, (unsigned long) 0, &msg_data,
                               &cred, sched, &laddr, &haddr, "KCVSV1.0");
        if (status != KSUCCESS)
            error (1, 0, "kerberos authentication failed: %s",
                   krb_get_err_text (status));
        memcpy (kblock, cred.session, sizeof (C_Block));
        return 1; /* success */
    }
4100 #endif /* HAVE_KERBEROS */


    static int
    send_variable_proc (node, closure)
        Node *node;
        void *closure;
    {
        send_to_server ("Set ", 0);
4110    send_to_server (node->key, 0);
        send_to_server ("=", 1);
        send_to_server (node->data, 0);
        send_to_server ("\012", 1);
        return 0;
    }

    /* Contact the server.  */
    void
    start_server ()
4120 {
        int tofd, fromfd;
        char *log = getenv ("CVS_CLIENT_LOG");

        /* Note that generally speaking we do *not* fall back to a different
           way of connecting if the first one does not work.  This is slow
           (*really* slow on a 14.4kbps link); the clean way to have a CVS
           which supports several ways of connecting is with access methods.  */

        /* Read in the line which lists the auth methods */
4130

        switch (CVSroot_method)
        {

    #ifdef AUTH_CLIENT_SUPPORT
        case pserver_method:
            /* Toss the return value.  It will die with error if anything
               goes wrong anyway. */
```

```
            connect_to_server (&tofd, &fromfd, 0, AUTH_PASSWORD);
            break;
#endif

#if HAVE_KERBEROS
        case kserver_method:
            connect_to_server (&tofd, &fromfd, 0, AUTH_KERBEROS_V4);
            /* This causes the server to send the greeting twice */
            /*     start_tcp_server (&tofd, &fromfd); */
            break;
#endif

#if HAVE_GSSAPI
            connect_to_server (&tofd, &fromfd, 0, AUTH_GSSAPI);
            break;
#endif

        case ext_method:
#if defined (NO_EXT_METHOD)
            error (0, 0, ":ext: method not supported by this port of CVS");
            error (1, 0, "try :server: instead");
#else
            start_rsh_server (&tofd, &fromfd);
#endif
            break;

        case server_method:
#if defined(START_SERVER)
            START_SERVER (&tofd, &fromfd, getcaller (),
                          CVSroot_username, CVSroot_hostname,
                          CVSroot_directory);
# if defined (START_SERVER_RETURNS_SOCKET) && defined (NO_SOCKET_TO_FD)
            /* This is a system on which we can only write to a socket
               using send/recv.  Therefore its START_SERVER needs to
               return a socket.  */
            use_socket_style = 1;
            server_sock = tofd;
# endif

#else
            /* FIXME: It should be possible to implement this portably,
               like pserver, which would get rid of the duplicated code
               in {vms,windows-NT,...}/startserver.c.  */
            error (1, 0, "\
the :server: access method is not supported by this port of CVS");
#endif
            break;

        default:
            error (1, 0, "\
(start_server internal error): unknown access method");
            break;
    }

    /* "Hi, I'm Darlene and I'll be your server tonight..." */
    server_started = 1;

#ifdef NO_SOCKET_TO_FD
    if (use_socket_style)
    {
        to_server = socket_buffer_initialize (server_sock, 0,
                                              (BUFMEMERRPROC) NULL);
        from_server = socket_buffer_initialize (server_sock, 1,
                                                (BUFMEMERRPROC) NULL);
    }
    else
#endif /* NO_SOCKET_TO_FD */
    {
        /* todo: some OS's don't need these calls... */
        close_on_exec (tofd);
        close_on_exec (fromfd);

        /* SCO 3 and AIX have a nasty bug in the I/O libraries which precludes
           fdopening the same file descriptor twice, so dup it if it is the
           same.  */
        if (tofd == fromfd)
        {
            fromfd = dup (tofd);
            if (fromfd < 0)
                error (1, errno, "cannot dup net connection");
        }

        /* These will use binary mode on systems which have it.  */
        to_server_fp = fdopen (tofd, FOPEN_BINARY_WRITE);
        if (to_server_fp == NULL)
            error (1, errno, "cannot fdopen %d for write", tofd);
        to_server = stdio_buffer_initialize (to_server_fp, 0,
                                             (BUFMEMERRPROC) NULL);

        from_server_fp = fdopen (fromfd, FOPEN_BINARY_READ);
```

```
    if (from_server_fp == NULL)
4230    error (1, errno, "cannot fdopen %d for read", fromfd);
    from_server = stdio_buffer_initialize (from_server_fp, 1,
                                           (BUFMEMERRPROC) NULL);
}

/* Set up logfiles, if any. */
if (log)
{
    int len = strlen (log);
    char *buf = xmalloc (len + 5);
4240    char *p;
    FILE *fp;

    strcpy (buf, log);
    p = buf + len;

    /* Open logfiles in binary mode so that they reflect
       exactly what was transmitted and received (that is
       more important than that they be maximally
       convenient to view). */
4250    /* Note that if we create several connections in a single CVS client
       (currently used by update.c), then the last set of logfiles will
       overwrite the others.  There is currently no way around this. */
    strcpy (p, ".in");
    fp = open_file (buf, "wb");
    if (fp == NULL)
        error (0, errno, "opening to-server logfile %s", buf);
    else
        to_server = log_buffer_initialize (to_server, fp, 0,
                                           (BUFMEMERRPROC) NULL);
4260
    strcpy (p, ".out");
    fp = open_file (buf, "wb");
    if (fp == NULL)
        error (0, errno, "opening from-server logfile %s", buf);
    else
        from_server = log_buffer_initialize (from_server, fp, 1,
                                             (BUFMEMERRPROC) NULL);

    free (buf);
4270}

/* Clear static variables. */
if (toplevel_repos != NULL)
    free (toplevel_repos);
toplevel_repos = NULL;
if (last_dir_name != NULL)
    free (last_dir_name);
last_dir_name = NULL;
if (last_repos != NULL)
4280    free (last_repos);
last_repos = NULL;
if (last_update_dir != NULL)
    free (last_update_dir);
last_update_dir = NULL;
stored_checksum_valid = 0;
stored_mode_valid = 0;

if (strcmp (command_name, "init") != 0)
{
4290    send_to_server ("Root ", 0);
    send_to_server (CVSroot_directory, 0);
    send_to_server ("\012", 1);
}

{
    struct response *rs;

    send_to_server ("Valid-responses", 0);

4300    for (rs = responses; rs->name != NULL; ++rs)
    {
        send_to_server (" ", 0);
        send_to_server (rs->name, 0);
    }
    send_to_server ("\012", 1);
}
send_to_server ("valid-requests\012", 0);

if (get_server_responses ())
4310    error_exit ();

/*
 * Now handle global options.
 *
 * -H, -f, -d, -e should be handled OK locally.
 *
 * -b we ignore (treating it as a server installation issue).
 * FIXME: should be an error message.
```

```
     *
     * -v we print local version info; FIXME: Add a protocol request to get
     * the version from the server so we can print that too.
     *
     * -l -t -r -w -q -n and -Q need to go to the server.
     */

    {
        int have_global = supported_request ("Global_option");

        if (noexec)
        {
            if (have_global)
            {
                send_to_server ("Global_option -n\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -n option.");
        }
        if (quiet)
        {
            if (have_global)
            {
                send_to_server ("Global_option -q\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -q option.");
        }
        if (really_quiet)
        {
            if (have_global)
            {
                send_to_server ("Global_option -Q\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -Q option.");
        }
        if (!cvswrite)
        {
            if (have_global)
            {
                send_to_server ("Global_option -r\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -r option.");
        }
        if (trace)
        {
            if (have_global)
            {
                send_to_server ("Global_option -t\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -t option.");
        }
        if (logoff)
        {
            if (have_global)
            {
                send_to_server ("Global_option -l\012", 0);
            }
            else
                error (1, 0,
                       "This server does not support the global -l option.");
        }
    }

    /* Find out about server-side cvswrappers.  An extra network
       turnaround for cvs import seems to be unavoidable, unless we
       want to add some kind of client-side place to configure which
       filenames imply binary.  For cvs add, we could avoid the
       problem by keeping a copy of the wrappers in CVSADM (the main
       reason to bother would be so we could make add work without
       contacting the server, I suspect).  */

    if ((strcmp (command_name, "import") == 0)
        || (strcmp (command_name, "add") == 0))
    {
        if (supported_request ("wrapper-sendme-rcsOptions"))
        {
            int err;
            send_to_server ("wrapper-sendme-rcsOptions\012", 0);
            err = get_server_responses ();
            if (err != 0)
                error (err, 0, "error reading from server");
```

```
        }
    }

    if (cvsencrypt)
    {
#ifdef ENCRYPTION
        /* Turn on encryption before turning on compression.  We do
           not want to try to compress the encrypted stream.  Instead,
           we want to encrypt the compressed stream.  If we can't turn
           on encryption, bomb out; don't let the user think the data
           is being encrypted when it is not.  */
#ifdef HAVE_KERBEROS
        if (CVSroot_method == kserver_method)
        {
            if (! supported_request ("Kerberos-encrypt"))
                error (1, 0, "This server does not support encryption");
            send_to_server ("Kerberos-encrypt\012", 0);
            to_server = krb_encrypt_buffer_initialize (to_server, 0, sched,
                                                       kblock,
                                                       (BUFMEMERRPROC) NULL);
            from_server = krb_encrypt_buffer_initialize (from_server, 1,
                                                         sched, kblock,
                                                         (BUFMEMERRPROC) NULL);
        }
        else
#endif /* HAVE_KERBEROS */
#ifdef HAVE_GSSAPI
        if (CVSroot_method == gserver_method)
        {
            if (! supported_request ("Gssapi-encrypt"))
                error (1, 0, "This server does not support encryption");
            send_to_server ("Gssapi-encrypt\012", 0);
            to_server = cvs_gssapi_wrap_buffer_initialize (to_server, 0,
                                                           gcontext,
                                                           ((BUFMEMERRPROC)
                                                            NULL));
            from_server = cvs_gssapi_wrap_buffer_initialize (from_server, 1,
                                                             gcontext,
                                                             ((BUFMEMERRPROC)
                                                              NULL));
            cvs_gssapi_encrypt = 1;
        }
        else
#endif /* HAVE_GSSAPI */
            error (1, 0, "Encryption is only supported when using GSSAPI or Kerberos");
#else /* ! ENCRYPTION */
        error (1, 0, "This client does not support encryption");
#endif /* ! ENCRYPTION */
    }

    if (gzip_level)
    {
        if (supported_request ("Gzip-stream"))
        {
            char gzip_level_buf[5];
            send_to_server ("Gzip-stream ", 0);
            sprintf (gzip_level_buf, "%d", gzip_level);
            send_to_server (gzip_level_buf, 0);
            send_to_server ("\012", 1);

            /* All further communication with the server will be
               compressed.  */

            to_server = compress_buffer_initialize (to_server, 0, gzip_level,
                                                    (BUFMEMERRPROC) NULL);
            from_server = compress_buffer_initialize (from_server, 1,
                                                      gzip_level,
                                                      (BUFMEMERRPROC) NULL);
        }
#ifndef NO_CLIENT_GZIP_PROCESS
        else if (supported_request ("gzip-file-contents"))
        {
            char gzip_level_buf[5];
            send_to_server ("gzip-file-contents ", 0);
            sprintf (gzip_level_buf, "%d", gzip_level);
            send_to_server (gzip_level_buf, 0);

            send_to_server ("\012", 1);

            file_gzip_level = gzip_level;
        }
#endif
        else
        {
            fprintf (stderr, "server doesn't support gzip-file-contents\n");
            /* Setting gzip_level to 0 prevents us from giving the
               error twice if update has to contact the server again
               to fetch unpatchable files.  */
            gzip_level = 0;
        }
```

```
    }

    if (cvsauthenticate && ! cvsencrypt)
    {
        /* Turn on authentication after turning on compression, so
           that we can compress the authentication information.  We
           assume that encrypted data is always authenticated--the
           ability to decrypt the data stream is itself a form of
           authentication.  */
#ifdef HAVE_GSSAPI
        if (CVSroot_method == gserver_method)
        {
            if (! supported_request ("Gssapi-authenticate"))
                error (1, 0,
                       "This server does not support stream authentication");
            send_to_server ("Gssapi-authenticate\012", 0);
            to_server = cvs_gssapi_wrap_buffer_initialize (to_server, 0,
                                                           gcontext,
                                                           ((BUFMEMERRPROC)
                                                            NULL));
            from_server = cvs_gssapi_wrap_buffer_initialize (from_server, 1,
                                                             gcontext,
                                                             ((BUFMEMERRPROC)
                                                              NULL));
        }
        else
            error (1, 0, "Stream authentication is only supported when using GSSAPI");
#else /* ! HAVE_GSSAPI */
        error (1, 0, "This client does not support stream authentication");
#endif /* ! HAVE_GSSAPI */
    }

#ifdef FILENAMES_CASE_INSENSITIVE
    if (supported_request ("Case"))
        send_to_server ("Case\012", 0);
#endif

    /* If "Set" is not supported, just silently fail to send the variables.
       Users with an old server should get a useful error message when it
       fails to recognize the ${=foo} syntax.  This way if someone uses
       several servers, some of which are new and some old, they can still
       set user variables in their .cvsrc without trouble.  */
    if (supported_request ("Set"))
        walklist (variable_list, send_variable_proc, NULL);
}

#ifndef NO_EXT_METHOD

/* Contact the server by starting it with rsh.  */

/* Right now, we have two different definitions for this function,
   depending on whether we start the rsh server using popenRW or not.
   This isn't ideal, and the best thing would probably be to change
   the OS/2 port to be more like the regular Unix client (i.e., by
   implementing piped_child). . . but I'm doing something else at the
   moment, and wish to make only one change at a time.  -Karl */

#ifdef START_RSH_WITH_POPEN_RW

/* This is actually a crock - it's OS/2-specific, for no one else
   uses it.  If I get time, I want to make piped_child and all the
   other stuff in os2/run.c work right.  In the meantime, this gets us
   up and running, and that's most important. */

static void
start_rsh_server (tofdp, fromfdp)
    int *tofdp, *fromfdp;
{
    int pipes[2];

    /* If you're working through firewalls, you can set the
       CVS_RSH environment variable to a script which uses rsh to
       invoke another rsh on a proxy machine.  */
    char *cvs_rsh = getenv ("CVS_RSH");
    char *cvs_server = getenv ("CVS_SERVER");
    int i = 0;
    /* This needs to fit "rsh", "-b", "-l", "USER", "host",
       "cmd (w/ args)", and NULL.  We leave some room to grow. */
    char *rsh_argv[10];

    if (!cvs_rsh)
        /* People sometimes suggest or assume that this should default
           to "remsh" on systems like HPUX in which that is the
           system-supplied name for the rsh program.  However, that
           causes various problems (keep in mind that systems such as
           HPUX might have non-system-supplied versions of "rsh", like
           a Kerberized one, which one might want to use).  If we
           based the name on what is found in the PATH of the person
           who runs configure, that would make it harder to
           consistently produce the same result in the face of
```

```
           different people producing binary distributions.  If we
           based it on "remsh" always being the default for HPUX
           (e.g. based on uname), that might be slightly better but
           would require us to keep track of what the defaults are for
           each system type, and probably would cope poorly if the
           existence of remsh or rsh varies from OS version to OS
           version.  Therefore, it seems best to have the default
           remain "rsh", and tell HPUX users to specify remsh, for
           example in CVS_RSH or other such mechanisms to be devised,
           if that is what they want (the manual already tells them
           that).  */
        cvs_rsh = "rsh";
    if (!cvs_server)
        cvs_server = "cvs";

    /* The command line starts out with rsh. */
    rsh_argv[i++] = cvs_rsh;

#ifdef RSH_NEEDS_BINARY_FLAG
    /* "-b" for binary, under OS/2. */
    rsh_argv[i++] = "-b";
#endif /* RSH_NEEDS_BINARY_FLAG */

    /* Then we strcat more things on the end one by one. */
    if (CVSroot_username != NULL)
    {
        rsh_argv[i++] = "-l";
        rsh_argv[i++] = CVSroot_username;
    }

    rsh_argv[i++] = CVSroot_hostname;
    rsh_argv[i++] = cvs_server;
    rsh_argv[i++] = "server";

    /* Mark the end of the arg list. */
    rsh_argv[i]   = (char *) NULL;

    if (trace)
    {
        fprintf (stderr, " -> Starting server: ");
        putc ('\n', stderr);
    }

    /* Do the deed. */
    rsh_pid = popenRW (rsh_argv, pipes);
    if (rsh_pid < 0)
        error (1, errno, "cannot start server via rsh");

    /* Give caller the file descriptors. */
    *tofdp   = pipes[0];
    *fromfdp = pipes[1];
}

#else /* ! START_RSH_WITH_POPEN_RW */

static void
start_rsh_server (tofdp, fromfdp)
     int *tofdp;
     int *fromfdp;
{
    /* If you're working through firewalls, you can set the
       CVS_RSH environment variable to a script which uses rsh to
       invoke another rsh on a proxy machine.  */
    char *cvs_rsh = getenv ("CVS_RSH");
    char *cvs_server = getenv ("CVS_SERVER");
    char *command;

    if (!cvs_rsh)
        cvs_rsh = "rsh";
    if (!cvs_server)
        cvs_server = "cvs";

    /* Pass the command to rsh as a single string.  This shouldn't
       affect most rsh servers at all, and will pacify some buggy
       versions of rsh that grab switches out of the middle of the
       command (they're calling the GNU getopt routines incorrectly).  */
    command = xmalloc (strlen (cvs_server)
                       + strlen (CVSroot_directory)
                       + 50);

    /* If you are running a very old (Nov 3, 1994, before 1.5)
     * version of the server, you need to make sure that your .bashrc
     * on the server machine does not set CVSROOT to something
     * containing a colon (or better yet, upgrade the server).  */
    sprintf (command, "%s server", cvs_server);

    {
        char *argv[10];
        char **p = argv;
```

```
        *p++ = cvs_rsh;
        *p++ = CVSroot_hostname;

        /* If the login names differ between client and server
         * pass it on to rsh.
         */
        if (CVSroot_username != NULL)
        {
            *p++ = "-l";
            *p++ = CVSroot_username;
        }

        *p++ = command;
        *p++ = NULL;

        if (trace)
        {
            int i;

            fprintf (stderr, " -> Starting server: ");
            for (i = 0; argv[i]; i++)
                fprintf (stderr, "%s ", argv[i]);
            putc ('\n', stderr);
        }
        rsh_pid = piped_child (argv, tofdp, fromfdp);

        if (rsh_pid < 0)
            error (1, errno, "cannot start server via rsh");
    }
    free (command);
}

#endif /* START_RSH_WITH_POPEN_RW */

#endif /* NO_EXT_METHOD */



/* Send an argument STRING.  */
void
send_arg (string)
    char *string;
{
    char buf[1];
    char *p = string;

    send_to_server ("Argument ", 0);

    while (*p)
    {
        if (*p == '\n')
        {
            send_to_server ("\012Argumentx ", 0);
        }
        else
        {
            buf[0] = *p;
            send_to_server (buf, 1);
        }
        ++p;
    }
    send_to_server ("\012", 1);
}

static void send_modified PROTO ((char *, char *, Vers_TS *));

/* VERS->OPTIONS specifies whether the file is binary or not.  NOTE: BEFORE
   using any other fields of the struct vers, we would need to fix
   client_process_import_file to set them up.  */

static void
send_modified (file, short_pathname, vers)
    char *file;
    char *short_pathname;
    Vers_TS *vers;
{
    /* File was modified, send it.  */
    struct stat sb;
    int fd;
    char *buf;
    char *mode_string;
    size_t bufsize;
    int bin;

    if (trace)
        (void) fprintf (stderr, " -> Sending file `%s' to server\n", file);

    /* Don't think we can assume fstat exists.  */
    if ( CVS_STAT (file, &sb) < 0)
        error (1, errno, "reading %s", short_pathname);
```

```
4770    mode_string = mode_to_string (sb.st_mode);

        /* Beware: on systems using CRLF line termination conventions,
           the read and write functions will convert CRLF to LF, so the
           number of characters read is not the same as sb.st_size.  Text
           files should always be transmitted using the LF convention, so
           we don't want to disable this conversion.  */
        bufsize = sb.st_size;
        buf = xmalloc (bufsize);

4780    /* Is the file marked as containing binary data by the "-kb" flag?
           If so, make sure to open it in binary mode: */

        if (vers && vers->options)
          bin = !(strcmp (vers->options, "-kb"));
        else
          bin = 0;

    #ifdef BROKEN_READWRITE_CONVERSION
        if (!bin)
4790    {
            /* If only stdio, not open/write/etc., do text/binary
               conversion, use convert_file which can compensate
               (FIXME: we could just use stdio instead which would
               avoid the whole problem).  */
            char tfile[1024]; strcpy(tfile, file); strcat(tfile, ".CVSBFCTMP");
            convert_file (file, O_RDONLY,
                          tfile, O_WRONLY | O_CREAT | O_TRUNC | OPEN_BINARY);
            fd = CVS_OPEN (tfile, O_RDONLY | OPEN_BINARY);
            if (fd < 0)
4800            error (1, errno, "reading %s", short_pathname);
        }
        else
            fd = CVS_OPEN (file, O_RDONLY | OPEN_BINARY);
    #else
        fd = CVS_OPEN (file, O_RDONLY | (bin ? OPEN_BINARY : 0));
    #endif

        if (fd < 0)
            error (1, errno, "reading %s", short_pathname);
4810
        if (file_gzip_level && sb.st_size > 100)
        {
            size_t newsize = 0;

            read_and_gzip (fd, short_pathname, (unsigned char **)&buf,
                           &bufsize, &newsize,
                           file_gzip_level);

            if (close (fd) < 0)
4820            error (0, errno, "warning: can't close %s", short_pathname);

            {
              char tmp[80];

              send_to_server ("Modified ", 0);
              send_to_server (file, 0);
              send_to_server ("\012", 1);
              send_to_server (mode_string, 0);
              send_to_server ("\012z", 2);
4830          sprintf (tmp, "%lu\n", (unsigned long) newsize);
              send_to_server (tmp, 0);

              send_to_server (buf, newsize);
            }
        }
        else
        {
            int newsize;

4840        {
              char *bufp = buf;
              int len;

              /* FIXME: This is gross.  It assumes that we might read
                 less than st_size bytes (true on NT), but not more.
                 Instead of this we should just be reading a block of
                 data (e.g. 8192 bytes), writing it to the network, and
                 so on until EOF.  */
              while ((len = read (fd, bufp, (buf + sb.st_size) - bufp)) > 0)
4850            bufp += len;

              if (len < 0)
                  error (1, errno, "reading %s", short_pathname);

              newsize = bufp - buf;
            }
            if (close (fd) < 0)
              error (0, errno, "warning: can't close %s", short_pathname);
```

```
4860        {
              char tmp[80];

              send_to_server ("Modified ", 0);
              send_to_server (file, 0);
              send_to_server ("\012", 1);
              send_to_server (mode_string, 0);
              send_to_server ("\012", 1);
              sprintf (tmp, "%lu\012", (unsigned long) newsize);
              send_to_server (tmp, 0);
4870        }
    #ifdef BROKEN_READWRITE_CONVERSION
            if (!bin)
            {
                char tfile[1024]; strcpy(tfile, file); strcat(tfile, ".CVSBFCTMP");
                if (CVS_UNLINK (tfile) < 0)
                    error (0, errno, "warning: can't remove temp file %s", tfile);
            }
    #endif

4880        /*
             * Note that this only ends with a newline if the file ended with
             * one.
             */
            if (newsize > 0)
                send_to_server (buf, newsize);
        }
        free (buf);
        free (mode_string);
    }
4890
    /* The address of an instance of this structure is passed to
       send_fileproc, send_filesdoneproc, and send_direntproc, as the
       callerdat parameter.  */

    struct send_data
    {
        /* Each of the following flags are zero for clear or nonzero for set.  */
        int build_dirs;
        int force;
4900    int no_contents;
    };

    static int send_fileproc PROTO ((void *callerdat, struct file_info *finfo));

    /* Deal with one file.  */
    static int
    send_fileproc (callerdat, finfo)
        void *callerdat;
        struct file_info *finfo;
4910 {
        struct send_data *args = (struct send_data *) callerdat;
        Vers_TS *vers;
        struct file_info xfinfo;
        /* File name to actually use.  Might differ in case from
           finfo->file.  */
        char *filename;

        send_a_repository ("", finfo->repository, finfo->update_dir);

4920    xfinfo = *finfo;
        xfinfo.repository = NULL;
        xfinfo.rcs = NULL;
        vers = Version_TS (&xfinfo, NULL, NULL, NULL, 0, 0);

        if (vers->entdata != NULL)
            filename = vers->entdata->user;
        else
            filename = finfo->file;

4930    if (vers->vn_user != NULL)
        {
            /* The Entries request.  */
            send_to_server ("Entry /", 0);
            send_to_server (filename, 0);
            send_to_server ("/", 0);
            send_to_server (vers->vn_user, 0);
            send_to_server ("/", 0);
            if (vers->ts_conflict != NULL)
            {
4940            if (vers->ts_user != NULL &&
                    strcmp (vers->ts_conflict, vers->ts_user) == 0)
                    send_to_server ("+=", 0);
                else
                    send_to_server ("+modified", 0);
            }
            send_to_server ("/", 0);
            send_to_server (vers->entdata != NULL
                            ? vers->entdata->options
```

```
                    : vers->options,
                    0);
        send_to_server ("/", 0);
        if (vers->entdata != NULL && vers->entdata->tag)
        {
            send_to_server ("T", 0);
            send_to_server (vers->entdata->tag, 0);
        }
        else if (vers->entdata != NULL && vers->entdata->date)
          {
            send_to_server ("D", 0);
            send_to_server (vers->entdata->date, 0);
          }
        send_to_server ("/", 0);
        send_to_server ("::", 0);
        send_to_server (CVSroot_directory, 0);
        send_to_server ("\012", 1);
    }
    else
    {
        /* It seems a little silly to re-read this on each file, but
           send_dirent_proc doesn't get called if filenames are specified
           explicitly on the command line.  */
        wrap_add_file (CVSDOTWRAPPER, 1);

        if (wrap_name_has (filename, WRAP_RCSOPTION))
        {
            /* No "Entry", but the wrappers did give us a kopt so we better
               send it with "Kopt".  As far as I know this only happens
               for "cvs add".  Question: is there any reason why checking
               for options from wrappers isn't done in Version_TS?

               Note: it might have been better to just remember all the
               kopts on the client side, rather than send them to the server,
               and have it send us back the same kopts.  But that seemed like
               a bigger change than I had in mind making now.  */

            if (supported_request ("Kopt"))
            {
                char *opt;

                send_to_server ("Kopt ", 0);
                opt = wrap_rcsoption (filename, 1);
                send_to_server (opt, 0);
                send_to_server ("\012", 1);
                free (opt);
            }
            else
                error (0, 0,
                       "\
warning: ignoring -k options due to server limitations");
        }
    }

    if (vers->ts_user == NULL)
    {
        /*
         * Do we want to print "file was lost" like normal CVS?
         * Would it always be appropriate?
         */
        /* File no longer exists.  Don't do anything, missing files
           just happen.  */
    }
    else if (vers->ts_rcs == NULL
             || args->force
             || strcmp (vers->ts_user, vers->ts_rcs) != 0)
    {
        if (args->no_contents
            && supported_request ("Is-modified"))
        {
            send_to_server ("Is-modified ", 0);
            send_to_server (filename, 0);
            send_to_server ("\012", 1);
        }
        else
            send_modified (filename, finfo->fullname, vers);
    }
    else
    {
        send_to_server ("Unchanged ", 0);
        send_to_server (filename, 0);
        send_to_server ("\012", 1);
    }

    /* if this directory has an ignore list, add this file to it */
    if (ignlist)
    {
        Node *p;

        p = getnode ();
```

```
        p->type = FILES;
        p->key = xstrdup (finfo->file);
        (void) addnode (ignlist, p);
      }

      if (handling_remotes) {
        /* Send the remote revisions needed for completion of this command */
        FILE* remote_rev_file = fopen (CVSADM_REMOTES, "r");
        char* line;
        int line_length;
        int line_chars_allocated;

        if (remote_rev_file != NULL) {
          FILE* new_remote_rev_file;
          rename (CVSADM_REMOTES, CVSADM_REMOTES_BACKUP);
          new_remote_rev_file = fopen (CVSADM_REMOTES, "w");
          if (new_remote_rev_file != NULL) {
            while ((line_length = getline (&line, &line_chars_allocated, remote_rev_file)) > 0) {
              char* file = NULL;
              char* rev = NULL;
              char* data = NULL;

              file = line;
              rev = strchr (file, '/');
              if (rev == NULL)
                continue;

              *rev = '\0';
              rev++;

              data = strchr (rev, '/');
              if (data == NULL)
                continue;

              *data = '\0';
              data++;

              *strchr (data, '\n') = '\0';

              if (strcmp (file, filename) == 0) {
                /* File matches */
                char* fullname = xmalloc (strlen (CVSADM) + strlen (data) + 5);
                sprintf (fullname, "%s/%s", CVSADM, data);
                send_remoterev (file, rev, fullname, vers);
                free (fullname);
              } else {
                fprintf (new_remote_rev_file, "%s\n", line);
              }
            }
            fclose (new_remote_rev_file);
            unlink (CVSADM_REMOTES_BACKUP);
          }
          fclose (remote_rev_file);
        }
      }
      freevers_ts (&vers);
      return 0;
    }

    int send_remoterev (char* file,
                        char* rev,
                        char* datafile,
                        Vers_TS* vers)
    {
      /* Send a remote revision to the server */
      struct stat sb;
      int fd;
      char *buf;
      char *mode_string;
      size_t bufsize;
      int bin;

      if (trace)
        (void) fprintf (stderr, " -> Sending remote revision of `%s' to server\n", file);

      /* Don't think we can assume fstat exists.  */
      if ( CVS_STAT (datafile, &sb) < 0)
        error (1, errno, "reading %s", datafile);

      mode_string = mode_to_string (sb.st_mode);

      /* Beware: on systems using CRLF line termination conventions,
         the read and write functions will convert CRLF to LF, so the
         number of characters read is not the same as sb.st_size.  Text
         files should always be transmitted using the LF convention, so
         we don't want to disable this conversion.  */
      bufsize = sb.st_size;
      buf = xmalloc (bufsize);

      /* Is the file marked as containing binary data by the "-kb" flag?
```

```
      If so, make sure to open it in binary mode: */

    if (vers && vers->options)
      bin = !(strcmp (vers->options, "-kb"));
    else
      bin = 0;

#ifdef BROKEN_READWRITE_CONVERSION
    if (!bin)
      {
        /* If only stdio, not open/write/etc., do text/binary
           conversion, use convert_file which can compensate
           (FIXME: we could just use stdio instead which would
           avoid the whole problem).  */
        char tfile[1024]; strcpy(tfile, datafile); strcat(tfile, ".CVSBFCTMP");
        convert_file (datafile, O_RDONLY,
                      tfile, O_WRONLY | O_CREAT | O_TRUNC | OPEN_BINARY);
        fd = CVS_OPEN (tfile, O_RDONLY | OPEN_BINARY);
        if (fd < 0)
          error (1, errno, "reading %s", short_pathname);
      }
    else
      fd = CVS_OPEN (datafile, O_RDONLY | OPEN_BINARY);
#else
    fd = CVS_OPEN (datafile, O_RDONLY | (bin ? OPEN_BINARY : 0));
#endif

    if (fd < 0)
      error (1, errno, "reading %s", datafile);

    if (file_gzip_level && sb.st_size > 100)
      {
        size_t newsize = 0;

        read_and_gzip (fd, datafile, (unsigned char **)&buf,
                       &bufsize, &newsize,
                       file_gzip_level);

        if (close (fd) < 0)
          error (0, errno, "warning: can't close %s", datafile);

        {
          char tmp[80];

          send_to_server ("Remote-revision ", 0);
          send_to_server (file, 0);
          send_to_server ("\012", 1);
          send_to_server (rev, 0);
          send_to_server ("\012", 1);
          send_to_server (mode_string, 0);
          send_to_server ("\012z", 2);
          sprintf (tmp, "%lu\n", (unsigned long) newsize);
          send_to_server (tmp, 0);

          send_to_server (buf, newsize);
        }
      }
    else
      {
        int newsize;

        {
          char *bufp = buf;
          int len;

          /* FIXME: This is gross.  It assumes that we might read
             less than st_size bytes (true on NT), but not more.
             Instead of this we should just be reading a block of
             data (e.g. 8192 bytes), writing it to the network, and
             so on until EOF.  */
          while ((len = read (fd, bufp, (buf + sb.st_size) - bufp)) > 0)
            bufp += len;

          if (len < 0)
            error (1, errno, "reading %s", datafile);

          newsize = bufp - buf;
        }
        if (close (fd) < 0)
          error (0, errno, "warning: can't close %s", datafile);

        {
          char tmp[80];

          send_to_server ("Remote-revision ", 0);
          send_to_server (file, 0);
          send_to_server ("\012", 1);
          send_to_server (rev, 0);
          send_to_server ("\012", 1);
          send_to_server (mode_string, 0);
```

```
        send_to_server ("\012", 1);
        sprintf (tmp, "%lu\n", (unsigned long) newsize);
        send_to_server (tmp, 0);
      }
#ifdef BROKEN_READWRITE_CONVERSION
      if (!bin)
        {
          char tfile[1024]; strcpy(tfile, datafile); strcat(tfile, ".CVSBFCTMP");
          if (CVS_UNLINK (tfile) < 0)
            error (0, errno, "warning: can't remove temp file %s", tfile);
        }
#endif

      /*
       * Note that this only ends with a newline if the file ended with
       * one.
       */
      if (newsize > 0)
        send_to_server (buf, newsize);
    }
  free (buf);
  free (mode_string);
}

static void send_ignproc PROTO ((char *, char *));

static void
send_ignproc (file, dir)
    char *file;
    char *dir;
{
    if (ign_inhibit_server || !supported_request ("Questionable"))
    {
        if (dir[0] != '\0')
            (void) printf ("? %s/%s\n", dir, file);
        else
            (void) printf ("? %s\n", file);
    }
    else
    {
        send_to_server ("Questionable ", 0);
        send_to_server (file, 0);
        send_to_server ("\012", 1);
    }
}

static int send_filesdoneproc PROTO ((void *, int, char *, char *, List *));

static int
send_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    /* if this directory has an ignore list, process it then free it */
    if (ignlist)
    {
        ignore_files (ignlist, entries, update_dir, send_ignproc);
        dellist (&ignlist);
    }

    return (err);
}

static Dtype send_dirent_proc PROTO ((void *, char *, char *, char *, List *));

/*
 * send_dirent_proc () is called back by the recursion processor before a
 * sub-directory is processed for update.
 * A return code of 0 indicates the directory should be
 * processed by the recursion code.  A return of non-zero indicates the
 * recursion code should skip this directory.
 *
 */
static Dtype
send_dirent_proc (callerdat, dir, repository, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repository;
    char *update_dir;
    List *entries;
{
    struct send_data *args = (struct send_data *) callerdat;
    int dir_exists;
    char *cvsadm_name;

    if (ignore_directory (update_dir))
    {
```

```
        /* print the warm fuzzy message */
5310    if (!quiet)
            error (0, 0, "Ignoring %s", update_dir);
        return (R_SKIP_ALL);
    }

    /*
     * If the directory does not exist yet (e.g. "cvs update -d foo"),
     * no need to send any files from it.  If the directory does not
     * have a CVS directory, then we pretend that it does not exist.
     * Otherwise, we will fail when trying to open the Entries file.
5320 * This case will happen when checking out a module defined as
     * "-a .".
     */
    cvsadm_name = xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
    sprintf (cvsadm_name, "%s/%s", dir, CVSADM);
    dir_exists = isdir (cvsadm_name);
    free (cvsadm_name);

    /* initialize the ignore list for this directory */
    ignlist = getlist ();
5330
    /*
     * If there is an empty directory (e.g. we are doing 'cvs add' on a
     * newly-created directory), the server still needs to know about it.
     */

    if (dir_exists)
    {
        /*
         * Get the repository from a CVS/Repository file whenever possible.
5340     * The repository variable is wrong if the names in the local
         * directory don't match the names in the repository.
         */
        char *repos = Name_Repository (dir, update_dir);
        send_a_repository (dir, repos, update_dir);
        free (repos);
    }
    else
    {
        /* It doesn't make sense to send a non-existent directory,
5350       because there is no way to get the correct value for
           the repository (I suppose maybe via the expand-modules
           request).  In the case where the "obvious" choice for
           repository is correct, the server can figure out whether
           to recreate the directory; in the case where it is wrong
           (that is, does not match what modules give us), we might as
           well just fail to recreate it.

           Checking for noexec is a kludge for "cvs -n add dir".  */
        /* Don't send a non-existent directory unless we are building
5360       new directories (build_dirs is true).  Otherwise, CVS may
           see a D line in an Entries file, and recreate a directory
           which the user removed by hand.  */
        if (args->build_dirs && noexec)
            send_a_repository (dir, repository, update_dir);
    }

    return (dir_exists ? R_PROCESS : R_SKIP_ALL);
}

5370 /*
 * Send each option in a string to the server, one by one.
 * This assumes that the options are separated by spaces, for example
 * STRING might be "-foo -C5 -y".
 */

void
send_option_string (string)
    char *string;
{
5380 char *copy;
    char *p;

    copy = xstrdup (string);
    p = copy;
    while (1)
    {
        char *s;
        char l;

5390    for (s = p; *s != ' ' && *s != '\0'; s++)
            ;
        l = *s;
        *s = '\0';
        if (s != p)
            send_arg (p);
        if (l == '\0')
            break;
        p = s + 1;
```

```
    }
    free (copy);
}


/* Send the names of all the argument files to the server.  */

void
send_file_names (argc, argv, flags)
    int argc;
    char **argv;
    unsigned int flags;
{
    int i;
    int level;
    int max_level;

    /* The fact that we do this here as well as start_recursion is a bit
       of a performance hit.  Perhaps worth cleaning up someday.  */
    if (flags & SEND_EXPAND_WILD)
        expand_wild (argc, argv, &argc, &argv);

    /* Send Max-dotdot if needed.  */
    max_level = 0;
    for (i = 0; i < argc; ++i)
    {
        level = pathname_levels (argv[i]);
        if (level > max_level)
            max_level = level;
    }
    if (max_level > 0)
    {
        if (supported_request ("Max-dotdot"))
        {
            char buf[10];
            sprintf (buf, "%d", max_level);

            send_to_server ("Max-dotdot ", 0);
            send_to_server (buf, 0);
            send_to_server ("\012", 1);
        }
        else
            /*
             * "leading .." is not strictly correct, as this also includes
             * cases like "foo/../..".  But trying to explain that in the
             * error message would probably just confuse users.
             */
            error (1, 0,
                   "leading .. not supported by old (pre-Max-dotdot) servers");
    }

    for (i = 0; i < argc; ++i)
    {
        char buf[1];
        char *p = argv[i];
        char *line = NULL;

#ifdef FILENAMES_CASE_INSENSITIVE
        /* We want to send the file name as it appears
           in CVS/Entries.  We put this inside an ifdef
           to avoid doing all these system calls in
           cases where fncmp is just strcmp anyway.  */
        /* For now just do this for files in the local
           directory.  Would be nice to handle the
           non-local case too, though.  */
        /* The isdir check could more gracefully be replaced
           with a way of having Entries_Open report back the
           error to us and letting us ignore existence_error.
           Or some such.  */
        if (p == last_component (p) && isdir (CVSADM))
        {
            List *entries;
            Node *node;

            /* If we were doing non-local directory,
               we would save_cwd, CVS_CHDIR
               like in update.c:isemptydir.  */
            /* Note that if we are adding a directory,
               the following will read the entry
               that we just wrote there, that is, we
               will get the case specified on the
               command line, not the case of the
               directory in the filesystem.  This
               is correct behavior.  */
            entries = Entries_Open (0, NULL);
            node = findnode_fn (entries, p);
            if (node != NULL)
            {
                line = xstrdup (node->key);
                p = line;
```

```
                delnode (node);
            }
            Entries_Close (entries);
        }
#endif /* FILENAMES_CASE_INSENSITIVE */

        send_to_server ("Argument ", 0);

        while (*p)
        {
            if (*p == '\n')
            {
                send_to_server ("\012Argumentx ", 0);
            }
            else if (ISDIRSEP (*p))
            {
                buf[0] = '/';
                send_to_server (buf, 1);
            }
            else
            {
                buf[0] = *p;
                send_to_server (buf, 1);
            }
            ++p;
        }
        send_to_server ("\012", 1);
        if (line != NULL)
            free (line);
    }

    if (flags & SEND_EXPAND_WILD)
    {
        int i;
        for (i = 0; i < argc; ++i)
            free (argv[i]);
        free (argv);
    }
}


/* Send Repository, Modified and Entry.  argc and argv contain only
   the files to operate on (or empty for everything), not options.
   local is nonzero if we should not recurse (-l option).  flags &
   SEND_BUILD_DIRS is nonzero if nonexistent directories should be
   sent.  flags & SEND_FORCE is nonzero if we should send unmodified
   files to the server as though they were modified.  flags &
   SEND_NO_CONTENTS means that this command only needs to know
   _whether_ a file is modified, not the contents.  Also sends Argument
   lines for argc and argv, so should be called after options are sent.  */
void
send_files (argc, argv, local, aflag, flags)
    int argc;
    char **argv;
    int local;
    int aflag;
    unsigned int flags;
{
    struct send_data args;
    int err;

    /*
     * aflag controls whether the tag/date is copied into the vers_ts.
     * But we don't actually use it, so I don't think it matters what we pass
     * for aflag here.
     */
    args.build_dirs = flags & SEND_BUILD_DIRS;
    args.force = flags & SEND_FORCE;
    args.no_contents = flags & SEND_NO_CONTENTS;
    err = start_recursion
        (send_fileproc, send_filesdoneproc,
         send_dirent_proc, (DIRLEAVEPROC)NULL, (void *) &args,
         argc, argv, local, W_LOCAL, aflag, 0, (char *)NULL, 0);
    if (err)
        error_exit ();
    if (toplevel_repos == NULL)
        /*
         * This happens if we are not processing any files,
         * or for checkouts in directories without any existing stuff
         * checked out.  The following assignment is correct for the
         * latter case; I don't think toplevel_repos matters for the
         * former.
         */
        toplevel_repos = xstrdup (CVSroot_directory);
    send_repository ("", toplevel_repos, ".");
}

void
client_import_setup (repository)
    char *repository;
```

```
{
    if (toplevel_repos == NULL) /* should always be true */
        send_a_repository ("", repository, "");
}

/*
 * Process the argument import file.
 */
int
client_process_import_file (message, vfile, vtag, targc, targv, repository,
                            all_files_binary)
    char *message;
    char *vfile;
    char *vtag;
    int targc;
    char *targv[];
    char *repository;
    int all_files_binary;
{
    char *update_dir;
    char *fullname;
    Vers_TS vers;

    assert (toplevel_repos != NULL);

    if (strncmp (repository, toplevel_repos, strlen (toplevel_repos)) != 0)
        error (1, 0,
               "internal error: pathname `%s' doesn't specify file in `%s'",
               repository, toplevel_repos);

    if (strcmp (repository, toplevel_repos) == 0)
    {
        update_dir = "";
        fullname = xstrdup (vfile);
    }
    else
    {
        update_dir = repository + strlen (toplevel_repos) + 1;

        fullname = xmalloc (strlen (vfile) + strlen (update_dir) + 10);
        strcpy (fullname, update_dir);
        strcat (fullname, "/");
        strcat (fullname, vfile);
    }

    send_a_repository ("", repository, update_dir);
    if (all_files_binary)
    {
        vers.options = xmalloc (4); /* strlen("-kb") + 1 */
        strcpy (vers.options, "-kb");
    }
    else
    {
        vers.options = wrap_rcsoption (vfile, 1);
    }
    if (vers.options != NULL)
    {
        if (supported_request ("Kopt"))
        {
            send_to_server ("Kopt ", 0);
            send_to_server (vers.options, 0);
            send_to_server ("\012", 1);
        }
        else
            error (0, 0,
                   "warning: ignoring -k options due to server limitations");
    }
    send_modified (vfile, fullname, &vers);
    if (vers.options != NULL)
        free (vers.options);
    free (fullname);
    return 0;
}

void
client_import_done ()
{
    if (toplevel_repos == NULL)
        /*
         * This happens if we are not processing any files,
         * or for checkouts in directories without any existing stuff
         * checked out.  The following assignment is correct for the
         * latter case; I don't think toplevel_repos matters for the
         * former.
         */
        /* FIXME: "can't happen" now that we call client_import_setup
           at the beginning.  */
        toplevel_repos = xstrdup (CVSroot_directory);
    send_repository ("", toplevel_repos, ".");
}
```

```
5670  static void
      notified_a_file (data, ent_list, short_pathname, filename)
          char *data;
          List *ent_list;
          char *short_pathname;
          char *filename;
      {
          FILE *fp;
          FILE *newf;
          size_t line_len = 8192;
5680      char *line = xmalloc (line_len);
          char *cp;
          int nread;
          int nwritten;
          char *p;

          fp = open_file (CVSADM_NOTIFY, "r");
          if (getline (&line, &line_len, fp) < 0)
          {
              if (feof (fp))
5690              error (0, 0, "cannot read %s: end of file", CVSADM_NOTIFY);
              else
                  error (0, errno, "cannot read %s", CVSADM_NOTIFY);
              goto error_exit;
          }
          cp = strchr (line, '\t');
          if (cp == NULL)
          {
              error (0, 0, "malformed %s file", CVSADM_NOTIFY);
              goto error_exit;
5700      }
          *cp = '\0';
          if (strcmp (filename, line + 1) != 0)
          {
              error (0, 0, "protocol error: notified %s, expected %s", filename,
                     line + 1);
          }

          if (getline (&line, &line_len, fp) < 0)
          {
5710          if (feof (fp))
              {
                  free (line);
                  if (fclose (fp) < 0)
                      error (0, errno, "cannot close %s", CVSADM_NOTIFY);
                  if ( CVS_UNLINK (CVSADM_NOTIFY) < 0)
                      error (0, errno, "cannot remove %s", CVSADM_NOTIFY);
                  return;
              }
              else
5720          {
                  error (0, errno, "cannot read %s", CVSADM_NOTIFY);
                  goto error_exit;
              }
          }
          newf = open_file (CVSADM_NOTIFYTMP, "w");
          if (fputs (line, newf) < 0)
          {
              error (0, errno, "cannot write %s", CVSADM_NOTIFYTMP);
              goto error2;
5730      }
          while ((nread = fread (line, 1, line_len, fp)) > 0)
          {
              p = line;
              while ((nwritten = fwrite (p, 1, nread, newf)) > 0)
              {
                  nread -= nwritten;
                  p += nwritten;
              }
              if (ferror (newf))
5740          {
                  error (0, errno, "cannot write %s", CVSADM_NOTIFYTMP);
                  goto error2;
              }
          }
          if (ferror (fp))
          {
              error (0, errno, "cannot read %s", CVSADM_NOTIFY);
              goto error2;
          }
5750      if (fclose (newf) < 0)
          {
              error (0, errno, "cannot close %s", CVSADM_NOTIFYTMP);
              goto error_exit;
          }
          free (line);
          if (fclose (fp) < 0)
          {
              error (0, errno, "cannot close %s", CVSADM_NOTIFY);
```

```
        return;
    }

    {
        /* In this case, we want rename_file() to ignore noexec. */
        int saved_noexec = noexec;
        noexec = 0;
        rename_file (CVSADM_NOTIFYTMP, CVSADM_NOTIFY);
        noexec = saved_noexec;
    }

    return;
  error2:
    (void) fclose (newf);
  error_exit:
    free (line);
    (void) fclose (fp);
}

static void
handle_notified (args, len)
    char *args;
    int len;
{
    call_in_directory (args, notified_a_file, NULL);
}

void
client_notify (repository, update_dir, filename, notif_type, val)
    char *repository;
    char *update_dir;
    char *filename;
    int notif_type;
    char *val;
{
    char buf[2];

    send_a_repository ("", repository, update_dir);
    send_to_server ("Notify ", 0);
    send_to_server (filename, 0);
    send_to_server ("\012", 1);
    buf[0] = notif_type;
    buf[1] = '\0';
    send_to_server (buf, 1);
    send_to_server ("\t", 1);
    send_to_server (val, 0);
}

/*
 * Send an option with an argument, dealing correctly with newlines in
 * the argument.  If ARG is NULL, forget the whole thing.
 */
void
option_with_arg (option, arg)
    char *option;
    char *arg;
{
    if (arg == NULL)
        return;

    send_to_server ("Argument ", 0);
    send_to_server (option, 0);
    send_to_server ("\012", 1);

    send_arg (arg);
}

/* Send a date to the server.  The input DATE is in RCS format.
   The time will be GMT.

   We then convert that to the format required in the protocol
   (including the "-D" option) and send it.  According to
   cvsclient.texi, RFC 822/1123 format is preferred, but for now we
   use the format that we always have, for
   conservatism/laziness/paranoia.  As far as I know all servers
   support the RFC 822/1123 format, so probably there would be no
   particular danger in switching.  */

void
client_senddate (date)
    const char *date;
{
    int year, month, day, hour, minute, second;
    char buf[100];

    if (sscanf (date, SDATEFORM, &year, &month, &day, &hour, &minute, &second)
        != 6)
    {
        error (1, 0, "client_senddate: sscanf failed on date");
    }
```

```
5850    sprintf (buf, "%d/%d/%d %d:%d:%d GMT", month, day, year,
            hour, minute, second);
    option_with_arg ("-D", buf);
}

void
send_init_command ()
{
    /* This is here because we need the CVSroot_directory variable.  */
    send_to_server ("init ", 0);
5860    send_to_server (CVSroot_directory, 0);
    send_to_server ("\012", 0);
}

typedef struct {
  char* repository;      /* Location of the file in the repository */
  char* server;          /* Name of the server */
  char* root;            /* Location of the repository on the server */
  int done;
  char* file;
5870  char* revision;
  int handling;
} remote_node;

typedef struct {
  int (*func) ();
  int argc;
  char** argv;
  int first_file_arg;
  int done;
5880 } process_remotes_closure;

static List* outstanding_remotes = NULL;
int first_file_arg;
int handling_remotes;
int fetch_remotes;

int process_remotes_func (Node* n, void* c);

enum {
5890  REMOTE_HANDLING_FETCH,
  REMOTE_HANDLING_RERUN,
  REMOTE_HANDLING_FETCH_WAITING,
  REMOTE_HANDLING_FETCHING,
  REMOTE_HANDLING_ADD_REMOTE_BRANCH
};

void add_remote (char* file, char* server, char* root, char* repository)
{
  remote_node* remote = malloc (sizeof (remote_node));
5900  Node* node = getnode ();
  node -> data = (char*) remote;

  if (outstanding_remotes == NULL) {
    outstanding_remotes = getlist ();
  }

  if (remote != NULL) {
    char* revision = strrchr (file, ' ') + 1;
    remote -> root = xstrdup (root);
5910    remote -> server = xstrdup (server);
    remote -> done = 0;
    remote -> file = xstrdup (file);
    remote -> revision = xstrdup (revision);
    remote -> file [revision - file - 1] = '\0';
    remote -> repository = xstrdup (repository);
    /* There are two things we may need to do with a remote: rerun the command
     * on a different server, or fetch the contents of the remote and rerun the
     * command on the original server. Fetching is used by commands which need
     * access to some revisions off a different server, i.e. diff and update -j
5920  */
    if (fetch_remotes) {
      remote -> handling = REMOTE_HANDLING_FETCH;
    } else {
      remote -> handling = REMOTE_HANDLING_RERUN;
    }

    addnode (outstanding_remotes, node);
  }
}
5930
void add_remote_tag (char* file, char* root, char* server, char* repository, char* revision)
{
  char* host = xstrdup (revision) + 1;
  char* path;
  char* rev;

  remote_node* remote = malloc (sizeof (remote_node));
  Node* node = getnode ();
```

```
    host = rev + 1;
    path = strchr (host, ':');
    if (path == NULL) {
      error (1, 1, "Invalid remote tag");
    } else {
      *path = '\0';
      path++;
    }

    rev = strchr (path, ':');
    if (rev == NULL) {
      error (1, 1, "Invalid remote tag");
    } else {
      *rev = '\0';
      rev++;
    }

    node -> data = (char*) remote;

    if (outstanding_remotes == NULL) {
      outstanding_remotes = getlist ();
    }

    if (remote != NULL) {
      remote -> root = xstrdup (path);
      remote -> server = xstrdup (host);
      remote -> done = 0;
      remote -> file = xstrdup (file);
      remote -> revision = xstrdup (rev);
      remote -> repository = repository;
      remote -> handling = REMOTE_HANDLING_ADD_REMOTE_BRANCH;

      addnode (outstanding_remotes, node);
    }
  }

  void add_remote_for_fetching (remote_node* old_remote)
  {
    remote_node* remote = malloc (sizeof (remote_node));
    Node* node = getnode ();
    node -> data = (char*) remote;

    if (outstanding_remotes == NULL) {
      outstanding_remotes = getlist ();
    }

    if (remote != NULL) {
      remote -> root = xstrdup (old_remote -> root);
      remote -> server = xstrdup (old_remote -> server);
      remote -> done = 0;
      remote -> file = xstrdup (old_remote -> file);
      remote -> revision = xstrdup (old_remote -> revision);
      remote -> repository = xstrdup (old_remote -> repository);
      remote -> handling = REMOTE_HANDLING_FETCHING;

      /* always add at front because we want them to be seen first, before their corresponding
         fetch entries */
      addnode_at_front (outstanding_remotes, node);
    }
  }

  int client_process_remotes (int (*func) (), int argc, char** argv)
  {
    /* Note that completed entries are not removed from the list. This
       is because there is no way to remove items from the list while
       iterating */
    process_remotes_closure closure = { func, argc, argv, 0, 1 };
    handling_remotes = 1;

    /* We dont' want any subsequently executed commands to affect this */
    closure.first_file_arg = first_file_arg;

    do {
      /* reset the fetch_remotes flag because the function we invoke
         to process remotes will set it as appropriate */
      fetch_remotes = 0;

      /* assume we are done */
      closure.done = 1;

      /* Always start with the initial arguments */
      closure.argc = argc;
      closure.argv = argv;
      closure.func = func;

      /* nuke the temporary file */
      unlink (CVSADM_REP_REMOTE);

      /* go through the list and figure out what to do with it */
```

```
    walklist (outstanding_remotes, process_remotes_func, &closure);

    /* If there are things not done, call the function (which
       may have been changed ) */
    if (!closure.done)
      (*(closure.func)) (closure.argc, closure.argv);
  } while (!closure.done);

  handling_remotes = 0;
  unlink (CVSADM_REP_REMOTE);

  return 0; /* we are done */
}

int remote_noop (int argc, char** argv)
{
  return 0;
}

int process_remotes_func (Node* n, void* c)
{
  process_remotes_closure* closure = c;

  remote_node* remote = (remote_node*) (n -> data);

  /* For now, each pass through the list only does one node, but that doesn't have to be
     the case in general */

  if (!closure -> done)
    return 0;

  /* Only process a node if it's still pending */

  if (!remote -> done) {
    switch (remote -> handling) {
    case REMOTE_HANDLING_RERUN: {
      /* If we are rerunning the commnd on a different server, setup the new arguments and root */
      int err = setup_root (remote);
      if (err == 0) {
        err = setup_args (closure, remote);
      }
      /* inform the caller that a command needs to be run */
      closure -> done = 0;
      /* Mark the node as done whether an error occurred or not - to avoid
         retrying. If we need to go to a different server to get the file,
         a new entry will appear in the remotes list anyway */
      remote -> done = 1;
      break;
    }

    case REMOTE_HANDLING_FETCH: {
      /* If we need to fetch a file before we can rerun the command, then we create a new entry
         in the remotes list, change the state of this entry, and move along. The next
         time through the list we will fetch the file and then we can retry this operation */
      remote -> handling = REMOTE_HANDLING_FETCH_WAITING;
      add_remote_for_fetching (remote);

      closure -> func = remote_noop;
      closure -> done = 0;
      break;
    }

    case REMOTE_HANDLING_FETCHING: {
      /* This is a file which needs to be fetched in order to complete some other operation in the
         list. We get it using update to stdout, but we send the output into a file in the CVS directory.
         After we are done, we find the corresponding "fetch" entry and let it know it can proceed */
      int err = setup_root (remote);
      if (err == 0) {
        err = setup_args_fetch (closure, remote);
        closure -> done = 0;
        remote -> done = 1;
      }
        break;
    }

    case REMOTE_HANDLING_FETCH_WAITING: {
      /* When we get to this node, the dependents have already been fetched, so we can just rerun
         the command. The command itself is responsible for sending the remote revisions to the server */
      int err = setup_root (remote);
      if (err == 0) {
        err = setup_args (closure, remote);
      }
      /* inform the caller that a command needs to be run */
      closure -> done = 0;
      /* Mark the node as done whether an error occurred or not - to avoid
         retrying. If we need to go to a different server to get the file,
         a new entry will appear in the remotes list anyway */
      remote -> done = 1;
      break;
    }
```

```
6120      case REMOTE_HANDLING_ADD_REMOTE_BRANCH: {
            /* When we are adding a remote branch, we need to create a file on the destination server
               which have one revision with exactly the same contents as the working files
               and make sure that that revision has the appropriate number and tag. To do that,
               the add command takes a new argument which allows the client to specify the remote
               branchpoint */
            int err = setup_root (remote);
            if(err == 0) {
              err = setup_args_remote_branch (closure, remote);
            }
6130        closure -> done = 0;
            remote -> done = 1;
            break;
          }
          }
        }
        return 0;
      }

      /* Change the CVS root in the global variables (grumble grumble) to
6140     match the CVS root needed for this remote file */
      int setup_root (remote_node* remote)
      {
        FILE* fpin;
        char* new_root = xmalloc (1 +
                                  strlen (method_names [CVSroot_method]) + 1 +
                                  ((CVSroot_username != NULL) ? strlen (CVSroot_username) : 0) + 1 +
                                  strlen (remote -> server) + 1 +
                                  strlen (remote -> root) + 10);
        sprintf (new_root, ":%s:%s%s:%s",
6150            method_names [CVSroot_method],
                (CVSroot_username != NULL) ? CVSroot_username : "",
                (CVSroot_username != NULL) ? "@" : "",
                remote -> server,
                remote -> root);

        parse_cvsroot (new_root);
        free (new_root);
        fpin = fopen (CVSADM_REP_REMOTE, "r");
        if (fpin == NULL) {
6160      fpin = fopen (CVSADM_REP_REMOTE, "w+");
          if (fpin != NULL) {
            fwrite (remote -> repository, 1, strlen (remote -> repository), fpin);
            fwrite ("\n", 1, 1, fpin);
            fclose (fpin);
          }
        } else {
          fclose (fpin);
        }
        return 0;
6170  }

      int setup_args (process_remotes_closure* closure, remote_node* remote)
      {
        int i;

        int newargc = closure -> first_file_arg;
        char** newargv = (char**) xmalloc (newargc * sizeof (char*));
        for (i = 0; i < newargc; i++) {
          if (i < closure -> first_file_arg) {
6180        newargv [i] = xstrdup (closure -> argv [i]);
          } else {
            newargv [i] = xstrdup (remote -> file);
          }
        }

        closure -> argc = newargc;
        closure -> argv = newargv;

        return 0;
6190  }

      int remote_fetch_update_wrapper (int argc, char** argv);
      int remote_branch_add_wrapper (int argc, char** argv);

      /* We always fetch a file with cvs update -p -r rev filename */
      int setup_args_fetch (process_remotes_closure* closure, remote_node* remote)
      {
        int i;
        char* args [5] = {"update", "-r", xstrdup (remote -> revision), xstrdup (remote -> file), NULL };
6200    closure -> func = &remote_fetch_update_wrapper;
        closure -> argc = 4;
        closure -> argv = (char**) xmalloc (sizeof (char*) * (closure -> argc + 1));
        for (i = 0; i <= closure -> argc; i++) {
          closure ->argv [i] = args [i];
        }
        return 0;
      }
```

```
int setup_args_remote_branch (process_remotes_closure* closure, remote_node* remote)
{
  int i;
  char* args [5] = {"add", "-r", xstrdup (remote -> revision), xstrdup (remote -> file), NULL };
  closure -> func = &remote_branch_add_wrapper;
  closure -> argc = 4;
  closure -> argv = (char**) xmalloc (sizeof (char*) * (closure -> argc + 1));
  for (i = 0; i <= closure -> argc; i++) {
    closure ->argv [i] = args [i];
  }
  return 0;
}

static int fetching_remotes = 0;

int fetch_remote (struct file_info* finfo)
{
  return 1;
}

int remote_fetch_update_wrapper (int argc, char** argv)
{
  int result;
  printf ("remote_fetch_update_wrapper %s %s %s %s\n", argv [0], argv [1], argv [2], argv [3]);
  fetching_remote = 1;
  result = update (argc, argv);
  fetching_remote = 0;
  return result;
}

int remote_branch_add_wrapper (int argc, char** argv)
{
  int result;
  printf ("remote_add_update_wrapper %s %s %s %s\n", argv [0], argv [1], argv [2], argv [3]);
  adding_remote = 1;
  result = add (argc, argv);
  adding_remote = 0;
  return result;
}

#endif /* CLIENT_SUPPORT */
```

A.9 client.h

```
/* Interface between the client and the rest of CVS.  */

/* Stuff shared with the server.  */
extern char *mode_to_string PROTO((mode_t));
extern int change_mode PROTO((char *, char *, int));

extern int gzip_level;
extern int file_gzip_level;
extern int filter_through_gzip PROTO((int, int, int, pid_t *));
extern int filter_through_gunzip PROTO((int, int, pid_t *));

#if defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT)

/* Whether the connection should be encrypted.  */
extern int cvsencrypt;

/* Whether the connection should use per-packet authentication.  */
extern int cvsauthenticate;

#ifdef ENCRYPTION

#ifdef HAVE_KERBEROS

/* We can't declare the arguments without including krb.h, and I don't
   want to do that in every file.  */
extern struct buffer *krb_encrypt_buffer_initialize ();

#endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI

/* Set this to turn on GSSAPI encryption.  */
extern int cvs_gssapi_encrypt;

#endif /* HAVE_GSSAPI */

#endif /* ENCRYPTION */

#ifdef HAVE_GSSAPI

/* We can't declare the arguments without including gssapi.h, and I
   don't want to do that in every file.  */
extern struct buffer *cvs_gssapi_wrap_buffer_initialize ();

#endif /* HAVE_GSSAPI */

#endif /* defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT) */

#ifdef CLIENT_SUPPORT
/*
 * Flag variable for seeing whether the server has been started yet.
 * As of this writing, only edit.c:notify_check() uses it.
 */
extern int server_started;

/* Is the -P option to checkout or update specified?  */
extern int client_prune_dirs;

#ifdef AUTH_CLIENT_SUPPORT
extern int use_authenticating_server;
void connect_to_server PROTO ((int *tofdp, int* fromfdp, int verify_only,
                               int method));
# ifndef CVS_AUTH_PORT
# define CVS_AUTH_PORT 24011
# endif /* CVS_AUTH_PORT */
#endif /* AUTH_CLIENT_SUPPORT */

#if defined (AUTH_SERVER_SUPPORT) || (defined (SERVER_SUPPORT) && \
defined (HAVE_GSSAPI)) || (SERVER_SUPPORT) && defined (HAVE_KERBEROS)
extern void server_authenticate_connection PROTO ((void));
#endif

/* Talking to the server. */
void send_to_server PROTO((char *str, size_t len));
void read_from_server PROTO((char *buf, size_t len));

/* Internal functions that handle client communication to server, etc.  */
int supported_request PROTO ((char *));
void option_with_arg PROTO((char *option, char *arg));

/* Get the responses and then close the connection.  */
extern int get_responses_and_close PROTO((void));

extern int get_server_responses PROTO((void));

/* Start up the connection to the server on the other end.  */
void
start_server PROTO((void));
```

```
90  /* Send the names of all the argument files to the server.  */
    void
    send_file_names PROTO((int argc, char **argv, unsigned int flags));

    /* Flags for send_file_names.  */
    /* Expand wild cards?  */
    #define SEND_EXPAND_WILD 1

    /*
     * Send Repository, Modified and Entry.  argc and argv contain only
100  * the files to operate on (or empty for everything), not options.
     * local is nonzero if we should not recurse (-l option).
     */
    void
    send_files PROTO((int argc, char **argv, int local, int aflag,
                      unsigned int flags));

    /* Flags for send_files.  */
    #define SEND_BUILD_DIRS 1
    #define SEND_FORCE 2
110 #define SEND_NO_CONTENTS 4

    /* Send an argument to the remote server.  */
    void
    send_arg PROTO((char *string));

    /* Send a string of single-char options to the remote server, one by one.  */
    void
    send_option_string PROTO((char *string));

120 extern void send_a_repository PROTO ((char *, char *, char *));

    #endif /* CLIENT_SUPPORT */

    /*
     * This structure is used to catalog the responses the client is
     * prepared to see from the server.
     */

    struct response
130 {
        /* Name of the response.  */
        char *name;

    #ifdef CLIENT_SUPPORT
        /*
         * Function to carry out the response.  ARGS is the text of the
         * command after name and, if present, a single space, have been
         * stripped off.  The function can scribble into ARGS if it wants.
         * Note that although LEN is given, ARGS is also guaranteed to be
140      * '\0' terminated.
         */
        void (*func) PROTO((char *args, int len));

        /*
         * ok and error are special; they indicate we are at the end of the
         * responses, and error indicates we should exit with nonzero
         * exitstatus.
         */
        enum {response_type_normal, response_type_ok, response_type_error} type;
150 #endif

        /* Used by the server to indicate whether response is supported by
           the client, as set by the Valid-responses request.  */
        enum {
          /*
           * Failure to implement this response can imply a fatal
           * error.  This should be set only for responses which were in the
           * original version of the protocol; it should not be set for new
           * responses.
160        */
          rs_essential,

          /* Some clients might not understand this response.  */
          rs_optional,

          /*
           * Set by the server to one of the following based on what this
           * client actually supports.
           */
170       rs_supported,
          rs_not_supported
          } status;
    };

    /* Table of responses ending in an entry with a NULL name.  */

    extern struct response responses[];
```

```
#ifdef CLIENT_SUPPORT
180
extern void client_senddate PROTO((const char *date));
extern void client_expand_modules PROTO((int argc, char **argv, int local));
extern void client_send_expansions PROTO((int local, char *where,
                                          int build_dirs));
extern void client_nonexpanded_setup PROTO((void));

extern int client_process_remotes (int (*func) (), int argc, char** argv);
extern int adding_remote;
struct file_info; // avoid circular includes
190 extern int fetch_remote (struct file_info*);

extern void send_init_command PROTO ((void));

extern char **failed_patches;
extern int failed_patches_count;
extern int tag_is_remote;
extern char *toplevel_wd;
extern void client_import_setup PROTO((char *repository));
extern int client_process_import_file
200     PROTO((char *message, char *vfile, char *vtag,
            int targc, char *targv[], char *repository, int all_files_binary));
extern void client_import_done PROTO((void));
extern void client_notify PROTO((char *, char *, char *, int, char *));
#endif /* CLIENT_SUPPORT */
```

A.10 commit.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Commit Files
 *
 * "commit" commits the present version to the RCS repository, AFTER
 * having done a test on conflicts.
 *
 * The call is: cvs commit [options] files...
 *
 */

#include <assert.h>
#include "cvs.h"
#include "getline.h"
#include "edit.h"
#include "fileattr.h"
#include "hardlink.h"

static Dtype check_direntproc PROTO ((void *callerdat, char *dir,
                                      char *repos, char *update_dir,
                                      List *entries));
static int check_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int check_filesdoneproc PROTO ((void *callerdat, int err,
                                       char *repos, char *update_dir,
                                       List *entries));
static int checkaddfile PROTO((char *file, char *repository, char *tag,
                               char *options, RCSNode **rcsnode));
static Dtype commit_direntproc PROTO ((void *callerdat, char *dir,
                                       char *repos, char *update_dir,
                                       List *entries));
static int commit_dirleaveproc PROTO ((void *callerdat, char *dir,
                                       int err, char *update_dir,
                                       List *entries));
static int commit_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int commit_filesdoneproc PROTO ((void *callerdat, int err,
                                        char *repository, char *update_dir,
                                        List *entries));
static int finaladd PROTO((struct file_info *finfo, char *revision, char *tag,
                           char *options));
static int findmaxrev PROTO((Node * p, void *closure));
static int lock_RCS PROTO((char *user, RCSNode *rcs, char *rev,
                           char *repository));
static int precommit_list_proc PROTO((Node * p, void *closure));
static int precommit_proc PROTO((char *repository, char *filter));
static int remove_file PROTO ((struct file_info *finfo, char *tag,
                               char *message));
static void fix_rcs_modes PROTO((char *rcs, char *user));
static void fixaddfile PROTO((char *file, char *repository));
static void fixbranch PROTO((RCSNode *, char *branch));
static void unlockrcs PROTO((RCSNode *rcs));
static void ci_delproc PROTO((Node *p));
static void masterlist_delproc PROTO((Node *p));
static char *locate_rcs PROTO((char *file, char *repository));

struct commit_info
{
    Ctype status;               /* as returned from Classify_File() */
    char *rev;                  /* a numeric rev, if we know it */
    char *tag;                  /* any sticky tag, or -r option */
    char *options;              /* Any sticky -k option */
};
struct master_lists
{
    List *ulist;                /* list for Update_Logfile */
    List *cilist;               /* list with commit_info structs */
};

static int force_ci = 0;
static int got_message;
static int run_module_prog = 1;
static int aflag;
static char *saved_tag;
static char *write_dirtag;
static int write_dirnonbranch;
static char *logfile;
static List *mulist;
static List *saved_ulist;
static char *saved_message;
static time_t last_register_time;

static const char *const commit_usage[] =
{
    "Usage: %s %s [-nRlf] [-m msg | -F logfile] [-r rev] files...\n",
```

```
    "\t-n\tDo not run the module program (if any).\n",
    "\t-R\tProcess directories recursively.\n",
    "\t-l\tLocal directory only (not recursive).\n",
    "\t-f\tForce the file to be committed; disables recursion.\n",
    "\t-F file\tRead the log message from file.\n",
    "\t-m msg\tLog message.\n",
    "\t-r rev\tCommit to this branch or trunk revision.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

#ifdef CLIENT_SUPPORT
/* Identify a file which needs "? foo" or a Questionable request.  */
struct question {
    /* The two fields for the Directory request.  */
    char *dir;
    char *repos;

    /* The file name.  */
    char *file;

    struct question *next;
};

struct find_data {
    List *ulist;
    int argc;
    char **argv;

    /* This is used from dirent to filesdone time, for each directory,
       to make a list of files we have already seen.  */
    List *ignlist;

    /* Linked list of files which need "? foo" or a Questionable request.  */
    struct question *questionables;

    /* Only good within functions called from the filesdoneproc.  Stores
       the repository (pointer into storage managed by the recursion
       processor.  */
    char *repository;

    /* Non-zero if we should force the commit.  This is enabled by
       either -f or -r options, unlike force_ci which is just -f.  */
    int force;
};

static Dtype find_dirent_proc PROTO ((void *callerdat, char *dir,
                                      char *repository, char *update_dir,
                                      List *entries));

static Dtype
find_dirent_proc (callerdat, dir, repository, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repository;
    char *update_dir;
    List *entries;
{
    struct find_data *find_data = (struct find_data *)callerdat;

    /* This check seems to slowly be creeping throughout CVS (update
       and send_dirent_proc by CVS 1.5, diff in 31 Oct 1995.  My guess
       is that it (or some variant thereof) should go in all the
       dirent procs.  Unless someone has some better idea...  */
    if (!isdir (dir))
        return (R_SKIP_ALL);

    /* initialize the ignore list for this directory */
    find_data->ignlist = getlist ();

    /* Print the same warm fuzzy as in check_direntproc, since that
       code will never be run during client/server operation and we
       want the messages to match. */
    if (!quiet)
        error (0, 0, "Examining %s", update_dir);

    return R_PROCESS;
}

/* Here as a static until we get around to fixing ignore_files to pass
   it along as an argument.  */
static struct find_data *find_data_static;

static void find_ignproc PROTO ((char *, char *));

static void
find_ignproc (file, dir)
    char *file;
    char *dir;
{
```

```
    struct question *p;

    p = (struct question *) xmalloc (sizeof (struct question));
    p->dir = xstrdup (dir);
    p->repos = xstrdup (find_data_static->repository);
    p->file = xstrdup (file);
    p->next = find_data_static->questionables;
    find_data_static->questionables = p;
}

static int find_filesdoneproc PROTO ((void *callerdat, int err,
                                      char *repository, char *update_dir,
                                      List *entries));

static int
find_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    struct find_data *find_data = (struct find_data *)callerdat;
    find_data->repository = repository;

    /* if this directory has an ignore list, process it then free it */
    if (find_data->ignlist)
    {
        find_data_static = find_data;
        ignore_files (find_data->ignlist, entries, update_dir, find_ignproc);
        dellist (&find_data->ignlist);
    }

    find_data->repository = NULL;

    return err;
}

static int find_fileproc PROTO ((void *callerdat, struct file_info *finfo));

/* Machinery to find out what is modified, added, and removed.  It is
   possible this should be broken out into a new client_classify function;
   merging it with classify_file is almost sure to be a mess, though,
   because classify_file has all kinds of repository processing.  */
static int
find_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    Vers_TS *vers;
    enum classify_type status;
    Node *node;
    struct find_data *args = (struct find_data *)callerdat;
    struct logfile_info *data;
    struct file_info xfinfo;

    /* if this directory has an ignore list, add this file to it */
    if (args->ignlist)
    {
        Node *p;

        p = getnode ();
        p->type = FILES;
        p->key = xstrdup (finfo->file);
        if (addnode (args->ignlist, p) != 0)
            freenode (p);
    }

    xfinfo = *finfo;
    xfinfo.repository = NULL;
    xfinfo.rcs = NULL;

    vers = Version_TS (&xfinfo, NULL, saved_tag, NULL, 0, 0);
    if (vers->ts_user == NULL
        && vers->vn_user != NULL
        && vers->vn_user[0] == '-')
        /* FIXME: If vn_user is starts with "-" but ts_user is
           non-NULL, what classify_file does is print "%s should be
           removed and is still there".  I'm not sure what it does
           then.  We probably should do the same.  */
        status = T_REMOVED;
    else if (vers->vn_user == NULL)
    {
        if (vers->ts_user == NULL)
            error (0, 0, "nothing known about `%s'", finfo->fullname);
        else
            error (0, 0, "use `%s add' to create an entry for %s",
                   program_name, finfo->fullname);
        return 1;
    }
```

```
    else if (vers->ts_user != NULL
             && vers->vn_user != NULL
             && vers->vn_user[0] == '0')
        /* FIXME: If vn_user is "0" but ts_user is NULL, what classify_file
           does is print "new-born %s has disappeared" and removes the entry.
           We probably should do the same.  */
        status = T_ADDED;
    else if (vers->ts_user != NULL
             && vers->ts_rcs != NULL
             && (args->force || strcmp (vers->ts_user, vers->ts_rcs) != 0))
        /* If we are forcing commits, pretend that the file is
           modified.  */
        status = T_MODIFIED;
    else
    {
        /* This covers unmodified files, as well as a variety of other
           cases.  FIXME: we probably should be printing a message and
           returning 1 for many of those cases (but I'm not sure
           exactly which ones).  */
        return 0;
    }

    node = getnode ();
    node->key = xstrdup (finfo->fullname);

    data = (struct logfile_info *) xmalloc (sizeof (struct logfile_info));
    data->type = status;
    data->tag = xstrdup (vers->tag);
    data->rev_old = data->rev_new = NULL;

    node->type = UPDATE;
    node->delproc = update_delproc;
    node->data = (char *) data;
    (void)addnode (args->ulist, node);

    ++args->argc;

    freevers_ts (&vers);
    return 0;
}

static int copy_ulist PROTO ((Node *, void *));

static int
copy_ulist (node, data)
    Node *node;
    void *data;
{
    struct find_data *args = (struct find_data *)data;
    args->argv[args->argc++] = node->key;
    return 0;
}
#endif /* CLIENT_SUPPORT */

int
commit (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int err = 0;
    int local = 0;

    if (argc == -1)
        usage (commit_usage);

#ifdef CVS_BADROOT
    /*
     * For log purposes, do not allow "root" to commit files.  If you look
     * like root, but are really logged in as a non-root user, it's OK.
     */
    /* FIXME: Shouldn't this check be much more closely related to the
       readonly user stuff (CVSROOT/readers, &c).  That is, why should
       root be able to "cvs init", "cvs import", &c, but not "cvs ci"?  */
    if (geteuid () == (uid_t) 0)
    {
        struct passwd *pw;

        if ((pw = (struct passwd *) getpwnam (getcaller ())) == NULL)
            error (1, 0, "you are unknown to this system");
        if (pw->pw_uid == (uid_t) 0)
            error (1, 0, "cannot commit files as 'root'");
    }
#endif /* CVS_BADROOT */

    optind = 0;
    while ((c = getopt (argc, argv, "+nlRm:fF:r:")) != -1)
    {
        switch (c)
        {
```

```
            case 'n':
                run_module_prog = 0;
                break;
            case 'm':
#ifdef FORCE_USE_EDITOR
                use_editor = 1;
#else
                use_editor = 0;
#endif
                if (saved_message)
                {
                    free (saved_message);
                    saved_message = NULL;
                }

                saved_message = xstrdup(optarg);
                break;
            case 'r':
                if (saved_tag)
                    free (saved_tag);
                saved_tag = xstrdup (optarg);
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'f':
                force_ci = 1;
                local = 1;              /* also disable recursion */
                break;
            case 'F':
#ifdef FORCE_USE_EDITOR
                use_editor = 1;
#else
                use_editor = 0;
#endif
                logfile = optarg;
                break;
            case '?':
            default:
                usage (commit_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    /* numeric specified revision means we ignore sticky tags... */
    if (saved_tag && isdigit (*saved_tag))
    {
        aflag = 1;
        /* strip trailing dots */
        while (saved_tag[strlen (saved_tag) - 1] == '.')
            saved_tag[strlen (saved_tag) - 1] = '\0';
    }

    /* some checks related to the "-F logfile" option */
    if (logfile)
    {
        int n, logfd;
        struct stat statbuf;

        if (saved_message)
            error (1, 0, "cannot specify both a message and a log file");

        /* FIXME: Why is this binary?  Needs more investigation.  */
        if ((logfd = CVS_OPEN (logfile, O_RDONLY | OPEN_BINARY)) < 0)
            error (1, errno, "cannot open log file %s", logfile);

        if (fstat(logfd, &statbuf) < 0)
            error (1, errno, "cannot find size of log file %s", logfile);

        saved_message = xmalloc (statbuf.st_size + 1);

        /* FIXME: Should keep reading until EOF, rather than assuming the
           first read gets the whole thing.  */
        if ((n = read (logfd, saved_message, statbuf.st_size + 1)) < 0)
            error (1, errno, "cannot read log message from %s", logfile);

        (void) close (logfd);
        saved_message[n] = '\0';
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        struct find_data find_args;
```

```
    ign_setup ();

    find_args.ulist = getlist ();
    find_args.argc = 0;
    find_args.questionables = NULL;
    find_args.ignlist = NULL;
    find_args.repository = NULL;

    /* It is possible that only a numeric tag should set this.
       I haven't really thought about it much.
       Anyway, I suspect that setting it unnecessarily only causes
       a little unneeded network traffic.  */
    find_args.force = force_ci || saved_tag != NULL;

    err = start_recursion (find_fileproc, find_filesdoneproc,
                           find_dirent_proc, (DIRLEAVEPROC) NULL,
                           (void *)&find_args,
                           argc, argv, local, W_LOCAL, 0, 0,
                           (char *)NULL, 0);
    if (err)
        error (1, 0, "correct above errors first!");

    if (find_args.argc == 0)
        /* Nothing to commit.  Exit now without contacting the
           server (note that this means that we won't print "?
           foo" for files which merit it, because we don't know
           what is in the CVSROOT/cvsignore file).  */
        return 0;

    /* Now we keep track of which files we actually are going to
       operate on, and only work with those files in the future.
       This saves time--we don't want to search the file system
       of the working directory twice.  */
    find_args.argv = (char **) xmalloc (find_args.argc * sizeof (char **));
    find_args.argc = 0;
    walklist (find_args.ulist, copy_ulist, &find_args);

    /* Do this before calling do_editor; don't ask for a log
       message if we can't talk to the server.  But do it after we
       have made the checks that we can locally (to more quickly
       catch syntax errors, the case where no files are modified,
       added or removed, etc.).

       On the other hand, calling start_server before do_editor
       means that we chew up server resources the whole time that
       the user has the editor open (hours or days if the user
       forgets about it), which seems dubious.  */
    start_server ();

    /*
     * We do this once, not once for each directory as in normal CVS.
     * The protocol is designed this way.  This is a feature.
     */
    if (use_editor)
        do_editor (".", &saved_message, (char *)NULL, find_args.ulist);

    /* Run the user-defined script to verify/check information in
     *the log message
     */
    do_verify (saved_message, (char *)NULL);

    /* We always send some sort of message, even if empty.  */
    /* FIXME: is that true?  There seems to be some code in do_editor
       which can leave the message NULL.  */
    option_with_arg ("-m", saved_message);

    /* OK, now process all the questionable files we have been saving
       up.  */
    {
        struct question *p;
        struct question *q;

        p = find_args.questionables;
        while (p != NULL)
        {
            if (ign_inhibit_server || !supported_request ("Questionable"))
            {
                cvs_output ("? ", 2);
                if (p->dir[0] != '\0')
                {
                    cvs_output (p->dir, 0);
                    cvs_output ("/", 1);
                }
                cvs_output (p->file, 0);
                cvs_output ("\n", 1);
            }
            else
            {
                send_to_server ("Directory ", 0);
                send_to_server (p->dir[0] == '\0' ? "." : p->dir, 0);
```

```
                send_to_server ("\012", 1);
                send_to_server (p->repos, 0);
                send_to_server ("\012", 1);

                send_to_server ("Questionable ", 0);
                send_to_server (p->file, 0);
                send_to_server ("\012", 1);
            }
            free (p->dir);
            free (p->repos);
            free (p->file);
            q = p->next;
            free (p);
            p = q;
        }
    }

    if (local)
        send_arg("-l");
    if (force_ci)
        send_arg("-f");
    if (!run_module_prog)
        send_arg("-n");
    option_with_arg ("-r", saved_tag);

    /* Sending only the names of the files which were modified, added,
       or removed means that the server will only do an up-to-date
       check on those files.  This is different from local CVS and
       previous versions of client/server CVS, but it probably is a Good
       Thing, or at least Not Such A Bad Thing.  */
    send_file_names (find_args.argc, find_args.argv, 0);

    /* FIXME: This whole find_args.force/SEND_FORCE business is a
       kludge.  It would seem to be a server bug that we have to
       say that files are modified when they are not.  This makes
       "cvs commit -r 2" across a whole bunch of files a very slow
       operation (and it isn't documented in cvsclient.texi).  I
       haven't looked at the server code carefully enough to be
       _sure_ why this is needed, but if it is because the "ci"
       program, which we used to call, wanted the file to exist,
       then it would be relatively simple to fix in the server.  */
    send_files (find_args.argc, find_args.argv, local, 0,
                find_args.force ? SEND_FORCE : 0);

    send_to_server ("ci\012", 0);
    err = get_responses_and_close ();
    if (err != 0 && use_editor && saved_message != NULL)
    {
        /* If there was an error, don't nuke the user's carefully
           constructed prose.  This is something of a kludge; a better
           solution is probably more along the lines of #150 in TODO
           (doing a second up-to-date check before accepting the
           log message has also been suggested, but that seems kind of
           iffy because the real up-to-date check could still fail,
           another error could occur, &c.  Also, a second check would
           slow things down).  */

        char *fname;
        FILE *fp;

        fname = cvs_temp_name ();
        fp = CVS_FOPEN (fname, "w+");
        if (fp == NULL)
            error (1, 0, "cannot create temporary file %s", fname);
        if (fwrite (saved_message, 1, strlen (saved_message), fp)
            != strlen (saved_message))
            error (1, errno, "cannot write temporary file %s", fname);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close temporary file %s", fname);
        error (0, 0, "saving log message in %s", fname);
    }
    return err;
}
#endif

    if (saved_tag != NULL)
        tag_check_valid (saved_tag, argc, argv, local, aflag, "");

    /* XXX - this is not the perfect check for this */
    if (argc <= 0)
        write_dirtag = saved_tag;

    wrap_setup ();

    lock_tree_for_write (argc, argv, local, aflag);

    /*
     * Set up the master update list and hard link list
     */
    mulist = getlist ();
```

```
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    if (preserve_perms)
    {
        hardlist = getlist ();

        /*
         * We need to save the working directory so that
         * check_fileproc can construct a full pathname for each file.
         */
        working_dir = xgetwd();
    }
#endif

    /*
     * Run the recursion processor to verify the files are all up-to-date
     */
    err = start_recursion (check_fileproc, check_filesdoneproc,
                           check_direntproc, (DIRLEAVEPROC) NULL, NULL, argc,
                           argv, local, W_LOCAL, aflag, 0, (char *) NULL, 1);
    if (err)
    {
        Lock_Cleanup ();
        error (1, 0, "correct above errors first!");
    }

    /*
     * Run the recursion processor to commit the files
     */
    write_dirnonbranch = 0;
    if (noexec == 0)
        err = start_recursion (commit_fileproc, commit_filesdoneproc,
                               commit_direntproc, commit_dirleaveproc, NULL,
                               argc, argv, local, W_LOCAL, aflag, 0,
                               (char *) NULL, 1);

    /*
     * Unlock all the dirs and clean up
     */
    Lock_Cleanup ();
    dellist (&mulist);

    if (last_register_time)
    {
        time_t now;

        (void) time (&now);
        if (now == last_register_time)
        {
            sleep (1);                  /* to avoid time-stamp races */
        }
    }

    return (err);
}

/* This routine determines the status of a given file and retrieves
   the version information that is associated with that file. */

static
Ctype
classify_file_internal (finfo, vers)
    struct file_info *finfo;
    Vers_TS **vers;
{
    int save_noexec, save_quiet, save_really_quiet;
    Ctype status;

    /* FIXME: Do we need to save quiet as well as really_quiet?  Last
       time I glanced at Classify_File I only saw it looking at really_quiet
       not quiet.  */
    save_noexec = noexec;
    save_quiet = quiet;
    save_really_quiet = really_quiet;
    noexec = quiet = really_quiet = 1;

    /* handle specified numeric revision specially */
    if (saved_tag && isdigit (*saved_tag))
    {
        /* If the tag is for the trunk, make sure we're at the head */
        if (numdots (saved_tag) < 2)
        {
            status = Classify_File (finfo, (char *) NULL, (char *) NULL,
                                    (char *) NULL, 1, aflag, vers, 0);
            if (status == T_UPTODATE || status == T_MODIFIED ||
                status == T_ADDED)
            {
                Ctype xstatus;

                freevers_ts (vers);
```

```
                xstatus = Classify_File (finfo, saved_tag, (char *) NULL,
                                         (char *) NULL, 1, aflag, vers, 0);
                if (xstatus == T_REMOVE_ENTRY)
                    status = T_MODIFIED;
                else if (status == T_MODIFIED && xstatus == T_CONFLICT)
                    status = T_MODIFIED;
                else
                    status = xstatus;
            }
        }
        else
        {
            char *xtag, *cp;

            /*
             * The revision is off the main trunk; make sure we're
             * up-to-date with the head of the specified branch.
             */
            xtag = xstrdup (saved_tag);
            if ((numdots (xtag) & 1) != 0)
            {
                cp = strrchr (xtag, '.');
                *cp = '\0';
            }
            status = Classify_File (finfo, xtag, (char *) NULL,
                                    (char *) NULL, 1, aflag, vers, 0);
            if ((status == T_REMOVE_ENTRY || status == T_CONFLICT)
                && (cp = strrchr (xtag, '.')) != NULL)
            {
                /* pluck one more dot off the revision */
                *cp = '\0';
                freevers_ts (vers);
                status = Classify_File (finfo, xtag, (char *) NULL,
                                        (char *) NULL, 1, aflag, vers, 0);
                if (status == T_UPTODATE || status == T_REMOVE_ENTRY)
                    status = T_MODIFIED;
            }
            /* now, muck with vers to make the tag correct */
            free ((*vers)->tag);
            (*vers)->tag = xstrdup (saved_tag);
            free (xtag);
        }
    }
    else
        status = Classify_File (finfo, saved_tag, (char *) NULL, (char *) NULL,
                                1, 0, vers, 0);
    noexec = save_noexec;
    quiet = save_quiet;
    really_quiet = save_really_quiet;

    return status;
}

/*
 * Check to see if a file is ok to commit and make sure all files are
 * up-to-date
 */
/* ARGSUSED */
static int
check_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    Ctype status;
    char *xdir;
    Node *p;
    List *ulist, *cilist;
    Vers_TS *vers;
    struct commit_info *ci;
    struct logfile_info *li;

    status = classify_file_internal (finfo, &vers);

    /*
     * If the force-commit option is enabled, and the file in question
     * appears to be up-to-date, just make it look modified so that
     * it will be committed.
     */
    if (force_ci && status == T_UPTODATE)
        status = T_MODIFIED;

    switch (status)
    {
        case T_CHECKOUT:
#ifdef SERVER_SUPPORT
        case T_PATCH:
#endif
        case T_NEEDS_MERGE:
        case T_CONFLICT:
        case T_REMOVE_ENTRY:
```

```
            error (0, 0, "Up-to-date check failed for `%s'", finfo->fullname);
            freevers_ts (&vers);
            return (1);
        case T_MODIFIED:
        case T_ADDED:
        case T_REMOVED:
            /*
             * some quick sanity checks; if no numeric -r option specified:
             * - can't have a sticky date
             * - can't have a sticky tag that is not a branch
             * Also,
             * - if status is T_REMOVED, can't have a numeric tag
             * - if status is T_ADDED, rcs file must not exist unless on
             *    a branch
             * - if status is T_ADDED, can't have a non-trunk numeric rev
             * - if status is T_MODIFIED and a Conflict marker exists, don't
             *    allow the commit if timestamp is identical or if we find
             *    an RCS_MERGE_PAT in the file.
             */
            if (!saved_tag || !isdigit (*saved_tag))
            {
                if (vers->date)
                {
                    error (0, 0,
                           "cannot commit with sticky date for file `%s'",
                           finfo->fullname);
                    freevers_ts (&vers);
                    return (1);
                }
                if (status == T_MODIFIED && vers->tag &&
                    !RCS_isbranch (finfo->rcs, vers->tag))
                {
                    error (0, 0,
                           "sticky tag `%s' for file `%s' is not a branch",
                           vers->tag, finfo->fullname);
                    freevers_ts (&vers);
                    return (1);
                }
            }
            if (status == T_MODIFIED && !force_ci && vers->ts_conflict)
            {
                char *filestamp;
                int retcode;

                /*
                 * We found a "conflict" marker.
                 *
                 * If the timestamp on the file is the same as the
                 * timestamp stored in the Entries file, we block the commit.
                 */
#ifdef SERVER_SUPPORT
                if (server_active)
                    retcode = vers->ts_conflict[0] != '=';
                else {
                    filestamp = time_stamp (finfo->file);
                    retcode = strcmp (vers->ts_conflict, filestamp);
                    free (filestamp);
                }
#else
                filestamp = time_stamp (finfo->file);
                retcode = strcmp (vers->ts_conflict, filestamp);
                free (filestamp);
#endif
                if (retcode == 0)
                {
                    error (0, 0,
                         "file `%s' had a conflict and has not been modified",
                           finfo->fullname);
                    freevers_ts (&vers);
                    return (1);
                }

                if (file_has_markers (finfo))
                {
                    /* Make this a warning, not an error, because we have
                       no way of knowing whether the "conflict indicators"
                       are really from a conflict or whether they are part
                       of the document itself (cvs.texinfo and sanity.sh in
                       CVS itself, for example, tend to want to have strings
                       like ">>>>>>>" at the start of a line).  Making people
                       kludge this the way they need to kludge keyword
                       expansion seems undesirable.  And it is worse than
                       keyword expansion, because there is no -ko
                       analogue.  */
                    error (0, 0,
                           "\
warning: file `%s' seems to still contain conflict indicators",
                           finfo->fullname);
                }
            }
```

```
900         if (status == T_REMOVED && vers->tag && isdigit (*vers->tag))
            {
                /* Remove also tries to forbid this, but we should check
                   here.  I'm only _sure_ about somewhat obscure cases
                   (hacking the Entries file, using an old version of
                   CVS for the remove and a new one for the commit), but
                   there might be other cases.  */
                error (0, 0,
          "cannot remove file `%s' which has a numeric sticky tag of `%s'",
                       finfo->fullname, vers->tag);
910             freevers_ts (&vers);
                return (1);
            }
            if (status == T_ADDED)
            {
                if (vers->tag == NULL)
                {
                    char *rcs;

                    rcs = xmalloc (strlen (finfo->repository)
920                                + strlen (finfo->file)
                                   + sizeof RCSEXT
                                   + 5);

                    /* Don't look in the attic; if it exists there we
                       will move it back out in checkaddfile.  */
                    sprintf(rcs, "%s/%s%s", finfo->repository, finfo->file,
                            RCSEXT);
                    if (isreadable (rcs))
                    {
930                     error (0, 0,
                    "cannot add file `%s' when RCS file `%s' already exists",
                             finfo->fullname, rcs);
                        freevers_ts (&vers);
                        free (rcs);
                        return (1);
                    }
                    free (rcs);
                }
                if (vers->tag && isdigit (*vers->tag) &&
940                 numdots (vers->tag) > 1)
                {
                    error (0, 0,
                "cannot add file `%s' with revision `%s'; must be on trunk",
                             finfo->fullname, vers->tag);
                    freevers_ts (&vers);
                    return (1);
                }
            }

950         /* done with consistency checks; now, to get on with the commit */
            if (finfo->update_dir[0] == '\0')
                xdir = ".";
            else
                xdir = finfo->update_dir;
            if ((p = findnode (mulist, xdir)) != NULL)
            {
                ulist = ((struct master_lists *) p->data)->ulist;
                cilist = ((struct master_lists *) p->data)->cilist;
            }
960         else
            {
                struct master_lists *ml;

                ulist = getlist ();
                cilist = getlist ();
                p = getnode ();
                p->key = xstrdup (xdir);
                p->type = UPDATE;
                ml = (struct master_lists *)
970                 xmalloc (sizeof (struct master_lists));
                ml->ulist = ulist;
                ml->cilist = cilist;
                p->data = (char *) ml;
                p->delproc = masterlist_delproc;
                (void) addnode (mulist, p);
            }

            /* first do ulist, then cilist */
            p = getnode ();
980         p->key = xstrdup (finfo->file);
            p->type = UPDATE;
            p->delproc = update_delproc;
            li = ((struct logfile_info *)
                  xmalloc (sizeof (struct logfile_info)));
            li->type = status;
            li->tag = xstrdup (vers->tag);
            li->rev_old = xstrdup (vers->vn_rcs);
            li->rev_new = NULL;
```

```
            p->data = (char *) li;
990         (void) addnode (ulist, p);

            p = getnode ();
            p->key = xstrdup (finfo->file);
            p->type = UPDATE;
            p->delproc = ci_delproc;
            ci = (struct commit_info *) xmalloc (sizeof (struct commit_info));
            ci->status = status;
            if (vers->tag)
                if (isdigit (*vers->tag))
1000                ci->rev = xstrdup (vers->tag);
                else
                    ci->rev = RCS_whatbranch (finfo->rcs, vers->tag);
            else
                ci->rev = (char *) NULL;
            ci->tag = xstrdup (vers->tag);
            ci->options = xstrdup(vers->options);
            p->data = (char *) ci;
            (void) addnode (cilist, p);

1010    #ifdef PRESERVE_PERMISSIONS_SUPPORT
            if (preserve_perms)
            {
                /* Add this file to hardlist, indexed on its inode.  When
                   we are done, we can find out what files are hardlinked
                   to a given file by looking up its inode in hardlist. */
                char *fullpath;
                Node *linkp;
                struct hardlink_info *hlinfo;

1020            /* Get the full pathname of the current file. */
                fullpath = xmalloc (strlen(working_dir) +
                                    strlen(finfo->fullname) + 2);
                sprintf (fullpath, "%s/%s", working_dir, finfo->fullname);

                /* To permit following links in subdirectories, files
                   are keyed on finfo->fullname, not on finfo->name. */
                linkp = lookup_file_by_inode (fullpath);

                /* If linkp is NULL, the file doesn't exist... maybe
1030               we're doing a remove operation? */
                if (linkp != NULL)
                {
                    /* Create a new hardlink_info node, which will record
                       the current file's status and the links listed in its
                       'hardlinks' delta field.  We will append this
                       hardlink_info node to the appropriate hardlist entry. */
                    hlinfo = (struct hardlink_info *)
                        xmalloc (sizeof (struct hardlink_info));
                    hlinfo->status = status;
1040                linkp->data = (char *) hlinfo;
                }
            }
        #endif

                break;
            case T_UNKNOWN:
                error (0, 0, "nothing known about `%s'", finfo->fullname);
                freevers_ts (&vers);
                return (1);
1050        case T_UPTODATE:
                break;
            default:
                error (0, 0, "CVS internal error: unknown status %d", status);
                break;
        }

        freevers_ts (&vers);
        return (0);
    }
1060
    /*
     * By default, return the code that tells do_recursion to examine all
     * directories
     */
    /* ARGSUSED */
    static Dtype
    check_direntproc (callerdat, dir, repos, update_dir, entries)
        void *callerdat;
        char *dir;
1070    char *repos;
        char *update_dir;
        List *entries;
    {
        if (!isdir (dir))
            return (R_SKIP_ALL);

        if (!quiet)
            error (0, 0, "Examining %s", update_dir);
```

```
    return (R_PROCESS);
}

/*
 * Walklist proc to run pre-commit checks
 */
static int
precommit_list_proc (p, closure)
    Node *p;
    void *closure;
{
    struct logfile_info *li;

    li = (struct logfile_info *) p->data;
    if (li->type == T_ADDED
        || li->type == T_MODIFIED
        || li->type == T_REMOVED)
    {
        run_arg (p->key);
    }
    return (0);
}

/*
 * Callback proc for pre-commit checking
 */
static int
precommit_proc (repository, filter)
    char *repository;
    char *filter;
{
    /* see if the filter is there, only if it's a full path */
    if (isabsolute (filter))
    {
        char *s, *cp;

        s = xstrdup (filter);
        for (cp = s; *cp; cp++)
            if (isspace (*cp))
            {
                *cp = '\0';
                break;
            }
        if (!isfile (s))
        {
            error (0, errno, "cannot find pre-commit filter `%s'", s);
            free (s);
            return (1);             /* so it fails! */
        }
        free (s);
    }

    run_setup (filter);
    run_arg (repository);
    (void) walklist (saved_ulist, precommit_list_proc, NULL);
    return (run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL|RUN_REALLY));
}

/*
 * Run the pre-commit checks for the dir
 */
/* ARGSUSED */
static int
check_filesdoneproc (callerdat, err, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
{
    int n;
    Node *p;

    /* find the update list for this dir */
    p = findnode (mulist, update_dir);
    if (p != NULL)
        saved_ulist = ((struct master_lists *) p->data)->ulist;
    else
        saved_ulist = (List *) NULL;

    /* skip the checks if there's nothing to do */
    if (saved_ulist == NULL || saved_ulist->list->next == saved_ulist->list)
        return (err);

    /* run any pre-commit checks */
    if ((n = Parse_Info (CVSROOTADM_COMMITINFO, repos, precommit_proc, 1)) > 0)
    {
        error (0, 0, "Pre-commit check failed");
        err += n;
```

```
    }

    return (err);
}

/*
 * Do the work of committing a file
 */
static int maxrev;
static char *sbranch;

/* ARGSUSED */
static int
commit_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    Node *p;
    int err = 0;
    List *ulist, *cilist;
    struct commit_info *ci;

    /* Keep track of whether write_dirtag is a branch tag.
       Note that if it is a branch tag in some files and a nonbranch tag
       in others, treat it as a nonbranch tag.  It is possible that case
       should elicit a warning or an error.  */
    if (write_dirtag != NULL
        && finfo->rcs != NULL)
    {
        char *rev = RCS_getversion (finfo->rcs, write_dirtag, NULL, 1, NULL);
        if (rev != NULL
            && !RCS_nodeisbranch (finfo->rcs, write_dirtag))
            write_dirnonbranch = 1;
        if (rev != NULL)
            free (rev);
    }

    if (finfo->update_dir[0] == '\0')
        p = findnode (mulist, ".");
    else
        p = findnode (mulist, finfo->update_dir);

    /*
     * if p is null, there were file type command line args which were
     * all up-to-date so nothing really needs to be done
     */
    if (p == NULL)
        return (0);
    ulist = ((struct master_lists *) p->data)->ulist;
    cilist = ((struct master_lists *) p->data)->cilist;

    /*
     * At this point, we should have the commit message unless we were called
     * with files as args from the command line.  In that latter case, we
     * need to get the commit message ourselves
     */
    if (!(got_message))
    {
        got_message = 1;
        if (use_editor)
            do_editor (finfo->update_dir, &saved_message,
                       finfo->repository, ulist);
        do_verify (saved_message, finfo->repository);
    }

    p = findnode (cilist, finfo->file);
    if (p == NULL)
        return (0);

    ci = (struct commit_info *) p->data;
    if (ci->status == T_MODIFIED)
    {
        if (finfo->rcs == NULL)
            error (1, 0, "internal error: no parsed RCS file");
        if (lock_RCS (finfo->file, finfo->rcs, ci->rev,
                      finfo->repository) != 0)
        {
            unlockrcs (finfo->rcs);
            err = 1;
            goto out;
        }
    }
    else if (ci->status == T_ADDED)
    {
        if (checkaddfile (finfo->file, finfo->repository, ci->tag, ci->options,
                          &finfo->rcs) != 0)
        {
            fixaddfile (finfo->file, finfo->repository);
            err = 1;
            goto out;
```

```
        }

        /* adding files with a tag, now means adding them on a branch.
           Since the branch test was done in check_fileproc for
           modified files, we need to stub it in again here. */

        if (ci->tag)
        {
            if (finfo->rcs == NULL)
                error (1, 0, "internal error: no parsed RCS file");
            ci->rev = RCS_whatbranch (finfo->rcs, ci->tag);
            err = Checkin ('A', finfo, finfo->rcs->path, ci->rev,
                           ci->tag, ci->options, saved_message);
            if (err != 0)
            {
                unlockrcs (finfo->rcs);
                fixbranch (finfo->rcs, sbranch);
            }

            (void) time (&last_register_time);

            ci->status = T_UPTODATE;
        }
    }

    /*
     * Add the file for real
     */
    if (ci->status == T_ADDED)
    {
        char *xrev = (char *) NULL;

        if (ci->rev == NULL)
        {
            /* find the max major rev number in this directory */
            maxrev = 0;
            (void) walklist (finfo->entries, findmaxrev, NULL);
            if (maxrev == 0)
                maxrev = 1;
            xrev = xmalloc (20);
            (void) sprintf (xrev, "%d", maxrev);
        }

        /* XXX - an added file with symbolic -r should add tag as well */
        err = finaladd (finfo, ci->rev ? ci->rev : xrev, ci->tag, ci->options);
        if (xrev)
            free (xrev);
    }
    else if (ci->status == T_MODIFIED)
    {
        err = Checkin ('M', finfo,
                       finfo->rcs->path, ci->rev, ci->tag,
                       ci->options, saved_message);

        (void) time (&last_register_time);

        if (err != 0)
        {
            unlockrcs (finfo->rcs);
            fixbranch (finfo->rcs, sbranch);
        }
    }
    else if (ci->status == T_REMOVED)
    {
        err = remove_file (finfo, ci->tag, saved_message);
#ifdef SERVER_SUPPORT
        if (server_active) {
            server_scratch_entry_only ();
            server_updated (finfo,
                            NULL,

                            /* Doesn't matter, it won't get checked. */
                            SERVER_UPDATED,

                            (mode_t) -1,
                            (unsigned char *) NULL,
                            (struct buffer *) NULL);
        }
#endif
    }

    /* Clearly this is right for T_MODIFIED.  I haven't thought so much
       about T_ADDED or T_REMOVED. */
    notify_do ('C', finfo->file, getcaller (), NULL, NULL, finfo->repository);

out:
    if (err != 0)
    {
        /* on failure, remove the file from ulist */
        p = findnode (ulist, finfo->file);
```

```
        if (p)
1350        delnode (p);
    }
    else
    {
        /* On success, retrieve the new version number of the file and
           copy it into the log information (see logmsg.c
           (logfile_write) for more details).  We should only update
           the version number for files that have been added or
           modified but not removed.  Why?  classify_file_internal
           will return the version number of a file even after it has
1360       been removed from the archive, which is not the behavior we
           want for our commitlog messages; we want the old version
           number and then "NONE." */

        if (ci->status != T_REMOVED)
        {
            p = findnode (ulist, finfo->file);
            if (p)
            {
                Vers_TS *vers;
1370            struct logfile_info *li;

                (void) classify_file_internal (finfo, &vers);
                li = (struct logfile_info *) p->data;
                li->rev_new = xstrdup (vers->vn_rcs);
                freevers_ts (&vers);
            }
        }
    }

1380    return (err);
}

/*
 * Log the commit and clean up the update list
 */
/* ARGSUSED */
static int
commit_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
1390    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    Node *p;
    List *ulist;

    p = findnode (mulist, update_dir);
    if (p == NULL)
1400    return (err);

    ulist = ((struct master_lists *) p->data)->ulist;

    got_message = 0;


    Update_Logfile (repository, saved_message, (FILE *) 0, ulist);

    /* Build the administrative files if necessary.  */
1410    {
        char *p;

        if (strncmp (CVSroot_directory, repository,
                     strlen (CVSroot_directory)) != 0)
            error (0, 0,
                "internal error: repository (%s) doesn't begin with root (%s)",
                repository, CVSroot_directory);
        p = repository + strlen (CVSroot_directory);
        if (*p == '/')
1420        ++p;
        if (strcmp ("CVSROOT", p) == 0
            /* Check for subdirectories because people may want to create
               subdirectories and list files therein in checkoutlist.  */
            || strncmp ("CVSROOT/", p, strlen ("CVSROOT/")) == 0
            )
        {
            /* "Database" might a little bit grandiose and/or vague,
               but "checked-out copies of administrative files, unless
               in the case of modules and you are using ndbm in which
1430           case modules.{pag,dir,db}" is verbose and excessively
               focused on how the database is implemented.  */

            /* mkmodules requires the absolute name of the CVSROOT directory.
               Remove anything after the 'CVSROOT' component - this is
               necessary when committing in a subdirectory of CVSROOT.  */
            char *admin_dir = xstrdup (repository);
            int cvsrootlen = strlen ("CVSROOT");
            assert (admin_dir[p - repository + cvsrootlen] == '\0'
```

```
                || admin_dir[p - repository + cvsrootlen] == '/');
1440        admin_dir[p - repository + cvsrootlen] = '\0';

            cvs_output (program_name, 0);
            cvs_output (" ", 1);
            cvs_output (command_name, 0);
            cvs_output (": Rebuilding administrative file database\n", 0);
            mkmodules (admin_dir);
            free (admin_dir);
        }
    }
1450
    if (err == 0 && run_module_prog)
    {
        FILE *fp;

        if ((fp = CVS_FOPEN (CVSADM_CIPROG, "r")) != NULL)
        {
            char *line;
            int line_length;
            size_t line_chars_allocated;
1460        char *repos;

            line = NULL;
            line_chars_allocated = 0;
            line_length = getline (&line, &line_chars_allocated, fp);
            if (line_length > 0)
            {
                /* Remove any trailing newline.  */
                if (line[line_length - 1] == '\n')
                    line[--line_length] = '\0';
1470            repos = Name_Repository ((char *) NULL, update_dir);
                run_setup (line);
                run_arg (repos);
                cvs_output (program_name, 0);
                cvs_output (" ", 1);
                cvs_output (command_name, 0);
                cvs_output (": Executing '", 0);
                run_print (stdout);
                cvs_output ("'\n", 0);
                (void) run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
1480            free (repos);
            }
            else
            {
                if (ferror (fp))
                    error (0, errno, "warning: error reading %s",
                           CVSADM_CIPROG);
            }
            if (line != NULL)
                free (line);
1490        if (fclose (fp) < 0)
                error (0, errno, "warning: cannot close %s", CVSADM_CIPROG);
        }
        else
        {
            if (! existence_error (errno))
                error (0, errno, "warning: cannot open %s", CVSADM_CIPROG);
        }
    }

1500    return (err);
}

/*
 * Get the log message for a dir
 */
/* ARGSUSED */
static Dtype
commit_direntproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
1510    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    Node *p;
    List *ulist;
    char *real_repos;

    if (!isdir (dir))
1520    return (R_SKIP_ALL);

    /* find the update list for this dir */
    p = findnode (mulist, update_dir);
    if (p != NULL)
        ulist = ((struct master_lists *) p->data)->ulist;
    else
        ulist = (List *) NULL;
```

```
    /* skip the files as an optimization */
    if (ulist == NULL || ulist->list->next == ulist->list)
        return (R_SKIP_FILES);

    /* get commit message */
    real_repos = Name_Repository (dir, update_dir);
    got_message = 1;
    if (use_editor)
        do_editor (update_dir, &saved_message, real_repos, ulist);
    do_verify (saved_message, real_repos);
    free (real_repos);
    return (R_PROCESS);
}

/*
 * Process the post-commit proc if necessary
 */
/* ARGSUSED */
static int
commit_dirleaveproc (callerdat, dir, err, update_dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update_dir;
    List *entries;
{
    /* update the per-directory tag info */
    /* FIXME?  Why?  The "commit examples" node of cvs.texinfo briefly
       mentions commit -r being sticky, but apparently in the context of
       this being a confusing feature!  */
    if (err == 0 && write_dirtag != NULL)
    {
        WriteTag (NULL, write_dirtag, NULL, write_dirnonbranch,
                  update_dir, Name_Repository (dir, update_dir));
    }

    return (err);
}

/*
 * find the maximum major rev number in an entries file
 */
static int
findmaxrev (p, closure)
    Node *p;
    void *closure;
{
    char *cp;
    int thisrev;
    Entnode *entdata;

    entdata = (Entnode *) p->data;
    if (entdata->type != ENT_FILE)
        return (0);
    cp = strchr (entdata->version, '.');
    if (cp != NULL)
        *cp = '\0';
    thisrev = atoi (entdata->version);
    if (cp != NULL)
        *cp = '.';
    if (thisrev > maxrev)
        maxrev = thisrev;
    return (0);
}

/*
 * Actually remove a file by moving it to the attic
 * XXX - if removing a ,v file that is a relative symbolic link to
 * another ,v file, we probably should add a ".." component to the
 * link to keep it relative after we move it into the attic.
 */
static int
remove_file (finfo, tag, message)
    struct file_info *finfo;
    char *tag;
    char *message;
{
    mode_t omask;
    int retcode;
    char *tmp;

    int branch;
    int lockflag;
    char *corev;
    char *rev;
    char *prev_rev;
    char *old_path;

    corev = NULL;
    rev = NULL;
```

```
    prev_rev = NULL;

    retcode = 0;

    if (finfo->rcs == NULL)
        error (1, 0, "internal error: no parsed RCS file");

    branch = 0;
    if (tag && !(branch = RCS_nodeisbranch (finfo->rcs, tag)))
    {
        /* a symbolic tag is specified; just remove the tag from the file */
        if ((retcode = RCS_deltag (finfo->rcs, tag)) != 0)
        {
            if (!quiet)
                error (0, retcode == -1 ? errno : 0,
                       "failed to remove tag `%s' from `%s'", tag,
                       finfo->fullname);
            return (1);
        }
        RCS_rewrite (finfo->rcs, NULL, NULL);
        Scratch_Entry (finfo->entries, finfo->file);
        return (0);
    }

    /* we are removing the file from either the head or a branch */
    /* commit a new, dead revision. */

    /* Print message indicating that file is going to be removed. */
    cvs_output ("Removing ", 0);
    cvs_output (finfo->fullname, 0);
    cvs_output (";\n", 0);

    rev = NULL;
    lockflag = 1;
    if (branch)
    {
        char *branchname;

        rev = RCS_whatbranch (finfo->rcs, tag);
        if (rev == NULL)
        {
            error (0, 0, "cannot find branch \"%s\".", tag);
            return (1);
        }

        branchname = RCS_getbranch (finfo->rcs, rev, 1);
        if (branchname == NULL)
        {
            /* no revision exists on this branch.  use the previous
               revision but do not lock. */
            corev = RCS_gettag (finfo->rcs, tag, 1, (int *) NULL);
            prev_rev = xstrdup(rev);
            lockflag = 0;
        } else
        {
            corev = xstrdup (rev);
            prev_rev = xstrdup(branchname);
            free (branchname);
        }

    } else  /* Not a branch */
    {
        /* Get current head revision of file. */
        prev_rev = RCS_head (finfo->rcs);
    }

    /* if removing without a tag or a branch, then make sure the default
       branch is the trunk. */
    if (!tag && !branch)
    {
        if (RCS_setbranch (finfo->rcs, NULL) != 0)
        {
            error (0, 0, "cannot change branch to default for %s",
                   finfo->fullname);
            return (1);
        }
        RCS_rewrite (finfo->rcs, NULL, NULL);
    }

#ifdef SERVER_SUPPORT
    if (server_active) {
        /* If this is the server, there will be a file sitting in the
           temp directory which is the kludgy way in which server.c
           tells time_stamp that the file is no longer around.  Remove
           it so we can create temp files with that name (ignore errors).  */
        unlink_file (finfo->file);
    }
#endif

    /* check something out.  Generally this is the head.  If we have a
```

```
        particular rev, then name it.  */
1710    retcode = RCS_checkout (finfo->rcs, finfo->file, rev ? corev : NULL,
                                (char *) NULL, (char *) NULL, RUN_TTY,
                                (RCSCHECKOUTPROC) NULL, (void *) NULL);
        if (retcode != 0)
        {
            error (0, 0,
                   "failed to check out `%s'", finfo->fullname);
            return (1);
        }

1720    /* Except when we are creating a branch, lock the revision so that
           we can check in the new revision.  */
        if (lockflag)
        {
            if (RCS_lock (finfo->rcs, rev ? corev : NULL, 1) == 0)
                RCS_rewrite (finfo->rcs, NULL, NULL);
        }

        if (corev != NULL)
            free (corev);
1730
        retcode = RCS_checkin (finfo->rcs, finfo->file, message, rev,
                               RCS_FLAGS_DEAD | RCS_FLAGS_QUIET);
        if (retcode != 0)
        {
            if (!quiet)
                error (0, retcode == -1 ? errno : 0,
                       "failed to commit dead revision for `%s'", finfo->fullname);
            return (1);
        }
1740
        if (rev != NULL)
            free (rev);

        old_path = finfo->rcs->path;
        if (!branch)
        {
            /* this was the head; really move it into the Attic */
            tmp = xmalloc(strlen(finfo->repository) +
                          sizeof('/') +
1750                      sizeof(CVSATTIC) +
                          sizeof('/') +
                          strlen(finfo->file) +
                          sizeof(RCSEXT) + 1);
            (void) sprintf (tmp, "%s/%s", finfo->repository, CVSATTIC);
            omask = umask (cvsumask);
            (void) CVS_MKDIR (tmp, 0777);
            (void) umask (omask);
            (void) sprintf (tmp, "%s/%s/%s%s", finfo->repository, CVSATTIC,
                            finfo->file, RCSEXT);
1760
            if (strcmp (finfo->rcs->path, tmp) != 0
                && CVS_RENAME (finfo->rcs->path, tmp) == -1
                && (isreadable (finfo->rcs->path) || !isreadable (tmp)))
            {
                free(tmp);
                return (1);
            }
            /* The old value of finfo->rcs->path is in old_path, and is
               freed below.  */
1770        finfo->rcs->path = tmp;
        }

        /* Print message that file was removed. */
        cvs_output (old_path, 0);
        cvs_output (" <-- ", 0);
        cvs_output (finfo->file, 0);
        cvs_output ("\nnew revision: delete; previous revision: ", 0);
        cvs_output (prev_rev, 0);
        cvs_output ("\ndone\n", 0);
1780    free(prev_rev);

        if (old_path != finfo->rcs->path)
            free (old_path);

        Scratch_Entry (finfo->entries, finfo->file);
        return (0);
    }

    /*
1790 * Do the actual checkin for added files
     */
    static int
    finaladd (finfo, rev, tag, options)
        struct file_info *finfo;
        char *rev;
        char *tag;
        char *options;
    {
```

```
    int ret;
    char *rcs;

    rcs = locate_rcs (finfo->file, finfo->repository);
    ret = Checkin ('A', finfo, rcs, rev, tag, options, saved_message);
    if (ret == 0)
    {
        char *tmp = xmalloc (strlen (finfo->file) + sizeof (CVSADM)
                             + sizeof (CVSEXT_LOG) + 10);
        (void) sprintf (tmp, "%s/%s%s", CVSADM, finfo->file, CVSEXT_LOG);
        (void) unlink_file (tmp);
        free (tmp);
    }
    else
        fixaddfile (finfo->file, finfo->repository);

    (void) time (&last_register_time);
    free (rcs);

    return (ret);
}

/*
 * Unlock an rcs file
 */
static void
unlockrcs (rcs)
    RCSNode *rcs;
{
    int retcode;

    if ((retcode = RCS_unlock (rcs, NULL, 0)) != 0)
        error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
               "could not unlock %s", rcs->path);
    else
        RCS_rewrite (rcs, NULL, NULL);
}

/*
 * remove a partially added file.  if we can parse it, leave it alone.
 */
static void
fixaddfile (file, repository)
    char *file;
    char *repository;
{
    RCSNode *rcsfile;
    char *rcs;
    int save_really_quiet;

    rcs = locate_rcs (file, repository);
    save_really_quiet = really_quiet;
    really_quiet = 1;
    if ((rcsfile = RCS_parsercsfile (rcs)) == NULL)
        (void) unlink_file (rcs);
    else
        freercsnode (&rcsfile);
    really_quiet = save_really_quiet;
    free (rcs);
}

/*
 * put the branch back on an rcs file
 */
static void
fixbranch (rcs, branch)
    RCSNode *rcs;
    char *branch;
{
    int retcode;

    if (branch != NULL)
    {
        if ((retcode = RCS_setbranch (rcs, branch)) != 0)
            error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
                   "cannot restore branch to %s for %s", branch, rcs->path);
        RCS_rewrite (rcs, NULL, NULL);
    }
}

/*
 * do the initial part of a file add for the named file.  if adding
 * with a tag, put the file in the Attic and point the symbolic tag
 * at the committed revision.
 */

static int
checkaddfile (file, repository, tag, options, rcsnode)
    char *file;
    char *repository;
```

```
    char *tag;
1890 char *options;
    RCSNode **rcsnode;
{
    char *rcs;
    char *fname;
    mode_t omask;
    int retcode = 0;
    int newfile = 0;
    RCSNode *rcsfile = NULL;
    int retval;
1900
    if (tag)
    {
        rcs = xmalloc (strlen (repository) + strlen (file)
                       + sizeof (RCSEXT) + sizeof (CVSATTIC) + 10);
        (void) sprintf (rcs, "%s/%s%s", repository, file, RCSEXT);
        if (! isreadable (rcs))
        {
            (void) sprintf(rcs, "%s/%s", repository, CVSATTIC);
            omask = umask (cvsumask);
1910        if (CVS_MKDIR (rcs, 0777) != 0 && errno != EEXIST)
                error (1, errno, "cannot make directory `%s'", rcs);;
            (void) umask (omask);
            (void) sprintf (rcs, "%s/%s/%s%s", repository, CVSATTIC, file,
                            RCSEXT);
        }
    }
    else
        rcs = locate_rcs (file, repository);

1920 if (isreadable (rcs))
    {
        /* file has existed in the past.  Prepare to resurrect. */
        char *rev;

        if ((rcsfile = *rcsnode) == NULL)
        {
            error (0, 0, "could not find parsed rcsfile %s", file);
            retval = 1;
            goto out;
1930    }

        if (tag == NULL)
        {
            char *oldfile;

            /* we are adding on the trunk, so move the file out of the
               Attic. */
            oldfile = xstrdup (rcs);
            sprintf (rcs, "%s/%s%s", repository, file, RCSEXT);
1940
            if (strcmp (oldfile, rcs) == 0)
            {
                error (0, 0, "internal error: confused about attic for %s",
                       oldfile);
            out1:
                free (oldfile);
                retval = 1;
                goto out;
            }
1950        if (CVS_RENAME (oldfile, rcs) != 0)
            {
                error (0, errno, "failed to move `%s' out of the attic",
                       oldfile);
                goto out1;
            }
            if (isreadable (oldfile)
                || !isreadable (rcs))
            {
                error (0, 0, "\
1960 internal error: `%s' didn't move out of the attic",
                       oldfile);
                goto out1;
            }
            free (oldfile);
            free (rcsfile->path);
            rcsfile->path = xstrdup (rcs);
        }

        rev = RCS_getversion (rcsfile, tag, NULL, 1, (int *) NULL);
1970    /* and lock it */
        if (lock_RCS (file, rcsfile, rev, repository))
        {
            error (0, 0, "cannot lock `%s'.", rcs);
            if (rev != NULL)
                free (rev);
            retval = 1;
            goto out;
        }
```

```
1980        if (rev != NULL)
                free (rev);
        }
        else
        {
            /* this is the first time we have ever seen this file; create
               an rcs file.  */

            char *desc;
            size_t descalloc;
1990        size_t desclen;

            char *opt;

            desc = NULL;
            descalloc = 0;
            desclen = 0;
            fname = xmalloc (strlen (file) + sizeof (CVSADM)
                             + sizeof (CVSEXT_LOG) + 10);
            (void) sprintf (fname, "%s/%s%s", CVSADM, file, CVSEXT_LOG);
2000        /* If the file does not exist, no big deal.  In particular, the
               server does not (yet at least) create CVSEXT_LOG files.  */
            if (isfile (fname))
                /* FIXME: Should be including update_dir in the appropriate
                   place here.  */
                get_file (fname, fname, "r", &desc, &descalloc, &desclen);
            free (fname);

            /* From reading the RCS 5.7 source, "rcs -i" adds a newline to the
               end of the log message if the message is nonempty.
2010           Do it.  RCS also deletes certain whitespace, in cleanlogmsg,
               which we don't try to do here.  */
            if (desclen > 0)
            {
                expand_string (&desc, &descalloc, desclen + 1);
                desc[desclen++] = '\012';
            }

            /* Set RCS keyword expansion options.  */
            if (options && options[0] == '-' && options[1] == 'k')
2020            opt = options + 2;
            else
                opt = NULL;

            /* This message is an artifact of the time when this
               was implemented via "rcs -i".  It should be revised at
               some point (does the "initial revision" in the message from
               RCS_checkin indicate that this is a new file?  Or does the
               "RCS file" message serve some function?).  */
            cvs_output ("RCS file: ", 0);
2030        cvs_output (rcs, 0);
            cvs_output ("\ndone\n", 0);

            if (add_rcs_file (NULL, rcs, file, NULL, opt,
                              NULL, NULL, 0, NULL,
                              desc, desclen, NULL, NULL) != 0)
            {
                retval = 1;
                goto out;
            }
2040        rcsfile = RCS_parsercsfile (rcs);
            newfile = 1;
            if (desc != NULL)
                free (desc);
        }

        /* when adding a file for the first time, and using a tag, we need
           to create a dead revision on the trunk.  */
        if (tag && newfile)
        {
2050        char *tmp;
            FILE *fp;

            /* move the new file out of the way. */
            fname = xmalloc (strlen (file) + sizeof (CVSADM)
                             + sizeof (CVSPREFIX) + 10);
            (void) sprintf (fname, "%s/%s%s", CVSADM, CVSPREFIX, file);
            rename_file (file, fname);

            /* Create empty FILE.  Can't use copy_file with a DEVNULL
2060           argument - copy_file now ignores device files. */
            fp = fopen (file, "w");
            if (fp == NULL)
                error (1, errno, "cannot open %s for writing", file);
            if (fclose (fp) < 0)
                error (0, errno, "cannot close %s", file);

            tmp = xmalloc (strlen (file) + strlen (tag) + 80);
            /* commit a dead revision. */
```

```
        (void) sprintf (tmp, "file %s was initially added on branch %s.",
                        file, tag);
        retcode = RCS_checkin (rcsfile, NULL, tmp, NULL,
                               RCS_FLAGS_DEAD | RCS_FLAGS_QUIET);
        free (tmp);
        if (retcode != 0)
        {
            error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
                   "could not create initial dead revision %s", rcs);
            retval = 1;
            goto out;
        }

        /* put the new file back where it was */
        rename_file (fname, file);
        free (fname);

        /* double-check that the file was written correctly */
        freercsnode (&rcsfile);
        rcsfile = RCS_parse (file, repository);
        if (rcsfile == NULL)
        {
            error (0, 0, "could not read %s", rcs);
            retval = 1;
            goto out;
        }
        if (rcsnode != NULL)
        {
            assert (*rcsnode == NULL);
            *rcsnode = rcsfile;
        }

        /* and lock it once again. */
        if (lock_RCS (file, rcsfile, NULL, repository))
        {
            error (0, 0, "cannot lock `%s'.", rcs);
            retval = 1;
            goto out;
        }
    }

    if (tag != NULL)
    {
        /* when adding with a tag, we need to stub a branch, if it
           doesn't already exist.  */

        if (rcsfile == NULL)
        {
            if (rcsnode != NULL && *rcsnode != NULL)
                rcsfile = *rcsnode;
            else
            {
                rcsfile = RCS_parse (file, repository);
                if (rcsfile == NULL)
                {
                    error (0, 0, "could not read %s", rcs);
                    retval = 1;
                    goto out;
                }
            }
        }

        if (!RCS_nodeisbranch (rcsfile, tag))
        {
            /* branch does not exist.  Stub it.  */
            char *head;
            char *magicrev;

            head = RCS_getversion (rcsfile, NULL, NULL, 0, (int *) NULL);
            magicrev = RCS_magicrev (rcsfile, head);

            retcode = RCS_settag (rcsfile, tag, magicrev);
            RCS_rewrite (rcsfile, NULL, NULL);

            free (head);
            free (magicrev);

            if (retcode != 0)
            {
                error (retcode == -1 ? 1 : 0, retcode == -1 ? errno : 0,
                       "could not stub branch %s for %s", tag, rcs);
                retval = 1;
                goto out;
            }
        }
        else
        {
            /* lock the branch. (stubbed branches need not be locked.)  */
            if (lock_RCS (file, rcsfile, NULL, repository))
            {
```

```
                error (0, 0, "cannot lock `%s'.", rcs);
2160            retval = 1;
                goto out;
            }
        }

        if (rcsnode && *rcsnode != rcsfile)
        {
            freercsnode(rcsnode);
            *rcsnode = rcsfile;
        }
2170    }

    fileattr_newfile (file);

    /* I don't think fix_rcs_modes is needed any more.  In the
       add_rcs_file case, the algorithms used by add_rcs_file and
       fix_rcs_modes are the same, so there is no need to go through
       it all twice.  In the other cases, I think we want to just
       preserve the mode that the file had before we started.  That is
       a behavior change, but I would think a desirable one.  */
2180    fix_rcs_modes (rcs, file);

    retval = 0;

 out:
    free (rcs);
    return retval;
}

/*
2190 * Attempt to place a lock on the RCS file; returns 0 if it could and 1 if it
 * couldn't.  If the RCS file currently has a branch as the head, we must
 * move the head back to the trunk before locking the file, and be sure to
 * put the branch back as the head if there are any errors.
 */
static int
lock_RCS (user, rcs, rev, repository)
    char *user;
    RCSNode *rcs;
    char *rev;
2200    char *repository;
{
    char *branch = NULL;
    int err = 0;

    /*
     * For a specified, numeric revision of the form "1" or "1.1", (or when
     * no revision is specified ""), definitely move the branch to the trunk
     * before locking the RCS file.
     *
2210 * The assumption is that if there is more than one revision on the trunk,
     * the head points to the trunk, not a branch... and as such, it's not
     * necessary to move the head in this case.
     */
    if (rev == NULL || (rev && isdigit (*rev) && numdots (rev) < 2))
    {
        branch = xstrdup (rcs->branch);
        if (branch != NULL)
        {
            if (RCS_setbranch (rcs, NULL) != 0)
2220        {
                error (0, 0, "cannot change branch to default for %s",
                       rcs->path);
                if (branch)
                    free (branch);
                return (1);
            }
        }
        err = RCS_lock(rcs, NULL, 1);
    }
2230    else
    {
        (void) RCS_lock(rcs, rev, 1);
    }

    /* We used to call RCS_rewrite here, and that might seem
       appropriate in order to write out the locked revision
       information.  However, such a call would actually serve no
       purpose.  CVS locks will prevent any interference from other
       CVS processes.  The comment above rcs_internal_lockfile
2240   explains that it is already unsafe to use RCS and CVS
       simultaneously.  It follows that writing out the locked
       revision information here would add no additional security.

       If we ever do care about it, the proper fix is to create the
       RCS lock file before calling this function, and maintain it
       until the checkin is complete.

       The call to RCS_lock is still required at present, since in
```

```
        some cases RCS_checkin will determine which revision to check
        in by looking for a lock.  FIXME: This is rather roundabout,
        and a more straightforward approach would probably be easier to
        understand.  */

    if (err == 0)
    {
        if (sbranch != NULL)
            free (sbranch);
        sbranch = branch;
        return (0);
    }

    /* try to restore the branch if we can on error */
    if (branch != NULL)
        fixbranch (rcs, branch);

    if (branch)
        free (branch);
    return (1);
}

/* Called when "add"ing files to the RCS respository.  It doesn't seem to
   be possible to get RCS to use the right mode, so we change it after
   the fact.  TODO: now that RCS has been librarified, we have the power
   to change this. */

static void
fix_rcs_modes (rcs, user)
    char *rcs;
    char *user;
{
    struct stat sb;
    mode_t rcs_mode;

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* Do ye nothing to the modes on a symbolic link. */
    if (preserve_perms && islink (user))
        return;
#endif

    if (CVS_STAT (user, &sb) < 0)
    {
        /* FIXME: Should be ->fullname.  */
        error (0, errno, "warning: cannot stat %s", user);
        return;
    }

    /* Now we compute the new mode.

       TODO: decide whether this whole thing can/should be skipped
       when 'preserve_perms' is set.  Almost certainly so. -twp

       The algorithm that we use is:

       Write permission is always off (this is what RCS and CVS have always
       done).

       If S_IRUSR is on (user read), then the read permission of
       the RCS file will be on.  It would seem that if this is off,
       then other users can't do "cvs update" and such, so perhaps this
       should be hardcoded to being on (it is a strange case, though--the
       case in which a user file doesn't have user read permission on).

       If S_IXUSR is on (user execute), then set execute permission
       on the RCS file.  This allows other users who check out the file
       to get the right setting for whether a shell script (for example)
       has the executable bit set.

       The result of that calculation is modified by CVSUMASK.  The
       reason, of course, that the read and execute settings take the
       user bit and copy it to all three bits (user, group, other), is
       that it should be CVSUMASK, not the umask of individual users,
       which is the sole determiner of modes in the repository.  */

    rcs_mode = 0;
    if (sb.st_mode & S_IRUSR)
        rcs_mode |= S_IRUSR | S_IRGRP | S_IROTH;
    if (sb.st_mode & S_IXUSR)
        rcs_mode |= S_IXUSR | S_IXGRP | S_IXOTH;
    rcs_mode &= ~cvsumask;
    if (chmod (rcs, rcs_mode) < 0)
        error (0, errno, "warning: cannot change mode of %s", rcs);
}

/*
 * free an UPDATE node's data
 */
void
update_delproc (p)
```

```
    Node *p;
{
    struct logfile_info *li;

    li = (struct logfile_info *) p->data;
    if (li->tag)
        free (li->tag);
    if (li->rev_old)
        free (li->rev_old);
    if (li->rev_new)
        free (li->rev_new);
    free (li);
}

/*
 * Free the commit_info structure in p.
 */
static void
ci_delproc (p)
    Node *p;
{
    struct commit_info *ci;

    ci = (struct commit_info *) p->data;
    if (ci->rev)
        free (ci->rev);
    if (ci->tag)
        free (ci->tag);
    if (ci->options)
        free (ci->options);
    free (ci);
}

/*
 * Free the commit_info structure in p.
 */
static void
masterlist_delproc (p)
    Node *p;
{
    struct master_lists *ml;

    ml = (struct master_lists *) p->data;
    dellist (&ml->ulist);
    dellist (&ml->cilist);
    free (ml);
}

/* Find an RCS file in the repository.  Most parts of CVS will want to
   rely instead on RCS_parse which performs a similar operation and is
   called by recurse.c which then puts the result in useful places
   like the rcs field of struct file_info.

   REPOSITORY is the repository (including the directory) and FILE is
   the filename within that directory (without RCSEXT).  Returns a
   newly-malloc'd array containing the absolute pathname of the RCS
   file that was found.  */
static char *
locate_rcs (file, repository)
    char *file;
    char *repository;
{
    char *rcs;

    rcs = xmalloc (strlen (repository) + strlen (file) + sizeof (RCSEXT) + 10);
    (void) sprintf (rcs, "%s/%s%s", repository, file, RCSEXT);
    if (!isreadable (rcs))
    {
        (void) sprintf (rcs, "%s/%s/%s%s", repository, CVSATTIC, file, RCSEXT);
        if (!isreadable (rcs))
            (void) sprintf (rcs, "%s/%s%s", repository, file, RCSEXT);
    }
    return rcs;
}
```

A.11 create_adm.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Create Administration.
 *
 * Creates a CVS administration directory based on the argument repository; the
 * "Entries" file is prefilled from the "initrecord" argument.
 */

#include "cvs.h"

/* update_dir includes dir as its last component.

   Return value is 0 for success, or 1 if we printed a warning message.
   Note that many errors are still fatal; particularly for unlikely errors
   a fatal error is probably better than a warning which might be missed
   or after which CVS might do something non-useful.  If WARN is zero, then
   don't print warnings; all errors are fatal then.  */

int
Create_Admin (dir, update_dir, repository, tag, date, nonbranch, warn)
    char *dir;
    char *update_dir;
    char *repository;
    char *tag;
    char *date;
    int nonbranch;
    int warn;
{
    FILE *fout;
    char *cp;
    char *reposcopy;
    char *tmp;

#ifdef SERVER_SUPPORT
    if (trace)
    {
        fprintf (stderr, "%c-> Create_Admin (%s, %s, %s, %s, %s, %d, %d)\n",
                 (server_active) ? 'S' : ' ',
                 dir, update_dir, repository, tag ? tag : "",
                 date ? date : "", nonbranch, warn);
    }
#endif

    if (noexec)
        return 0;

    tmp = xmalloc (strlen (dir) + 100);
    if (dir != NULL)
        (void) sprintf (tmp, "%s/%s", dir, CVSADM);
    else
        (void) strcpy (tmp, CVSADM);
    if (isfile (tmp))
        error (1, 0, "there is a version in %s already", update_dir);

    if (CVS_MKDIR (tmp, 0777) < 0)
    {
        /* We want to print out the entire update_dir, since a lot of
           our code calls this function with dir == "." or dir ==
           NULL.  I hope that gives enough information in cases like
           absolute pathnames; printing out xgetwd or something would
           be way too verbose in the common cases.  */

        if (warn)
        {
            /* The reason that this is a warning, rather than silently
               just skipping creating the directory, is that we don't want
               CVS's behavior to vary subtly based on factors (like directory
               permissions) which are not made clear to the user.  With
               the warning at least we let them know what is going on.  */
            error (0, errno, "warning: cannot make directory %s in %s",
                   CVSADM, update_dir);
            free (tmp);
            return 1;
        }
        else
            error (1, errno, "cannot make directory %s in %s",
                   CVSADM, update_dir);
    }

    /* record the current cvs root for later use */

    Create_Root (dir, CVSroot_original);
    if (dir != NULL)
```

```
            (void) sprintf (tmp, "%s/%s", dir, CVSADM_REP);
        else
            (void) strcpy (tmp, CVSADM_REP);
        fout = CVS_FOPEN (tmp, "w+");
        if (fout == NULL)
        {
            if (update_dir[0] == '\0')
                error (1, errno, "cannot open %s", tmp);
            else
                error (1, errno, "cannot open %s/%s", update_dir, CVSADM_REP);
        }
        reposcopy = xstrdup (repository);
        Sanitize_Repository_Name (reposcopy);

        /* The top level of the repository is a special case - we need to
           write it with an extra dot at the end.  This trailing '.' stuff
           rubs me the wrong way - on the other hand, I don't want to
           spend the time making sure all of the code can handle it if we
           don't do it. */

        if (strcmp (reposcopy, CVSroot_directory) == 0)
        {
            reposcopy = xrealloc (reposcopy, strlen (reposcopy) + 3);
            strcat (reposcopy, "/.");
        }

        cp = reposcopy;

    #ifdef RELATIVE_REPOS
        /*
         * If the Repository file is to hold a relative path, try to strip off
         * the leading CVSroot argument.
         */
        if (CVSroot_directory != NULL)
        {
            char *path = xmalloc (strlen (CVSroot_directory) + 10);

            (void) sprintf (path, "%s/", CVSroot_directory);
            if (strncmp (cp, path, strlen (path)) == 0)
                cp += strlen (path);
            free (path);
        }
    #endif

        if (fprintf (fout, "%s\n", cp) < 0)
        {
            if (update_dir[0] == '\0')
                error (1, errno, "write to %s failed", tmp);
            else
                error (1, errno, "write to %s/%s failed", update_dir, CVSADM_REP);
        }
        if (fclose (fout) == EOF)
        {
            if (update_dir[0] == '\0')
                error (1, errno, "cannot close %s", tmp);
            else
                error (1, errno, "cannot close %s/%s", update_dir, CVSADM_REP);
        }

        /* now, do the Entries file */
        if (dir != NULL)
            (void) sprintf (tmp, "%s/%s", dir, CVSADM_ENT);
        else
            (void) strcpy (tmp, CVSADM_ENT);
        fout = CVS_FOPEN (tmp, "w+");
        if (fout == NULL)
        {
            if (update_dir[0] == '\0')
                error (1, errno, "cannot open %s", tmp);
            else
                error (1, errno, "cannot open %s/%s", update_dir, CVSADM_ENT);
        }
        if (fclose (fout) == EOF)
        {
            if (update_dir[0] == '\0')
                error (1, errno, "cannot close %s", tmp);
            else
                error (1, errno, "cannot close %s/%s", update_dir, CVSADM_ENT);
        }

        /* Create a new CVS/Tag file */
        WriteTag (dir, tag, date, nonbranch, update_dir, repository);

    #ifdef SERVER_SUPPORT
        if (server_active)
        {
            server_template (update_dir, repository);
        }

        if (trace)
```

```
    {
180     fprintf (stderr, "%c<- Create_Admin\n",
                (server_active) ? 'S' : ' ');
    }
#endif

    free (reposcopy);
    free (tmp);
    return 0;
}
```

A.12 cvs.h

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS kit.
 */

/*
 * basic information used in all source files
 *
 */


#include "config.h"      /* this is stuff found via autoconf */
#include "options.h"     /* these are some larger questions which
                            can't easily be automatically checked
                            for */

/* Changed from if __STDC__ to ifdef __STDC__ because of Sun's acc compiler */

#ifdef __STDC__
#define PTR void *
#else
#define PTR char *
#endif

/* Add prototype support.  */
#ifndef PROTO
#if defined (USE_PROTOTYPES) ? USE_PROTOTYPES : defined (__STDC__)
#define PROTO(ARGS) ARGS
#else
#define PROTO(ARGS) ()
#endif
#endif

#include <stdio.h>

/* Under OS/2, <stdio.h> doesn't define popen()/pclose(). */
#ifdef USE_OWN_POPEN
#include "popen.h"
#endif

#ifdef STDC_HEADERS
#include <stdlib.h>
#else
extern void exit ();
extern char *getenv();
#endif

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif

#ifdef HAVE_STRING_H
#include <string.h>
#else
#include <strings.h>
#endif

#ifdef SERVER_SUPPORT
/* If the system doesn't provide strerror, it won't be declared in
   string.h.  */
char *strerror ();
#endif

#include <fnmatch.h> /* This is supposed to be available on Posix systems */

#include <ctype.h>
#include <pwd.h>
#include <signal.h>

#ifdef HAVE_ERRNO_H
#include <errno.h>
#else
#ifndef errno
extern int errno;
#endif /* !errno */
#endif /* HAVE_ERRNO_H */

#include "system.h"

#include "hash.h"
#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
#include "client.h"
#endif

#ifdef MY_NDBM
```

```
#include "myndbm.h"
#else
#include <ndbm.h>
#endif /* MY_NDBM */

#include "regex.h"
#include "getopt.h"
#include "wait.h"

#include "rcs.h"


/* This actually gets set in system.h.  Note that the _ONLY_ reason for
   this is if various system calls (getwd, getcwd, readlink) require/want
   us to use it.  All other parts of CVS allocate pathname buffers
   dynamically, and we want to keep it that way.  */
#ifndef PATH_MAX
#ifdef MAXPATHLEN
#define PATH_MAX MAXPATHLEN+2
#else
#define PATH_MAX 1024+2
#endif
#endif /* PATH_MAX */

/* Definitions for the CVS Administrative directory and the files it contains.
   Here as #define's to make changing the names a simple task.  */

#ifdef USE_VMS_FILENAMES
#define CVSADM          "CVS"
#define CVSADM_ENT      "CVS/Entries."
#define CVSADM_ENTBAK   "CVS/Entries.Backup"
#define CVSADM_ENTLOG   "CVS/Entries.Log"
#define CVSADM_ENTSTAT  "CVS/Entries.Static"
#define CVSADM_REP      "CVS/Repository."
#define CVSADM_REP_REMOTE      "CVS/Repository.remote"
#define CVSADM_REMOTE_TMP      "CVS/Remote"
#define CVSADM_ROOT     "CVS/Root."
#define CVSADM_CIPROG   "CVS/Checkin.prog"
#define CVSADM_UPROG    "CVS/Update.prog"
#define CVSADM_TAG      "CVS/Tag."
#define CVSADM_NOTIFY   "CVS/Notify."
#define CVSADM_NOTIFYTMP "CVS/Notify.tmp"
#define CVSADM_BASE     "CVS/Base"
#define CVSADM_BASEREV   "CVS/Baserev."
#define CVSADM_BASEREVTMP "CVS/Baserev.tmp"
#define CVSADM_TEMPLATE "CVS/Template."
#else /* USE_VMS_FILENAMES */
#define CVSADM "CVS"
#define CVSADM_ENT "CVS/Entries"
#define CVSADM_REMOTES  "CVS/Remotes"
#define CVSADM_REMOTES_BACKUP  "CVS/Remotes.Backup"
#define CVSADM_ENTBAK "CVS/Entries.Backup"
#define CVSADM_ENTLOG "CVS/Entries.Log"
#define CVSADM_ENTSTAT "CVS/Entries.Static"
#define CVSADM_REP "CVS/Repository"
#define CVSADM_REP_REMOTE      "CVS/Repository.remote"
#define CVSADM_REMOTE_TMP      "CVS/Remote"
#define CVSADM_ROOT "CVS/Root"
#define CVSADM_CIPROG "CVS/Checkin.prog"
#define CVSADM_UPROG "CVS/Update.prog"
#define CVSADM_TAG "CVS/Tag"
#define CVSADM_NOTIFY "CVS/Notify"
#define CVSADM_NOTIFYTMP "CVS/Notify.tmp"
/* A directory in which we store base versions of files we currently are
   editing with "cvs edit".  */
#define CVSADM_BASE     "CVS/Base"
#define CVSADM_BASEREV  "CVS/Baserev"
#define CVSADM_BASEREVTMP "CVS/Baserev.tmp"
/* File which contains the template for use in log messages.  */
#define CVSADM_TEMPLATE "CVS/Template"
#endif /* USE_VMS_FILENAMES */

#define CVSADM_CACHED_REMOTE_SEPARATOR ">"

/* This is the special directory which we use to store various extra
   per-directory information in the repository.  It must be the same as
   CVSADM to avoid creating a new reserved directory name which users cannot
   use, but is a separate #define because if anyone changes it (which I don't
   recommend), one needs to deal with old, unconverted, repositories.

   See fileattr.h for details about file attributes, the only thing stored
   in CVSREP currently.  */
#define CVSREP "CVS"

/*
 * Definitions for the CVSROOT Administrative directory and the files it
 * contains.  This directory is created as a sub-directory of the $CVSROOT
 * environment variable, and holds global administration information for the
 * entire source repository beginning at $CVSROOT.
 */
```

```
#define CVSROOTADM "CVSROOT"
#define CVSROOTADM_MODULES "modules"
#define CVSROOTADM_LOGINFO "loginfo"
#define CVSROOTADM_RCSINFO "rcsinfo"
#define CVSROOTADM_COMMITINFO "commitinfo"
#define CVSROOTADM_TAGINFO "taginfo"
#define CVSROOTADM_EDITINFO "editinfo"
#define CVSROOTADM_VERIFYMSG "verifymsg"
#define CVSROOTADM_HISTORY "history"
#define CVSROOTADM_VALTAGS "val-tags"
#define CVSROOTADM_IGNORE "cvsignore"
#define CVSROOTADM_CHECKOUTLIST "checkoutlist"
#define CVSROOTADM_WRAPPER "cvswrappers"
#define CVSROOTADM_NOTIFY "notify"
#define CVSROOTADM_USERS "users"
#define CVSROOTADM_READERS "readers"
#define CVSROOTADM_WRITERS "writers"
#define CVSROOTADM_PASSWD "passwd"
#define CVSROOTADM_CONFIG "config"

#define CVSNULLREPOS "Emptydir" /* an empty directory */

/* Other CVS file names */

/* Files go in the attic if the head main branch revision is dead,
   otherwise they go in the regular repository directories.  The whole
   concept of having an attic is sort of a relic from before death
   support but on the other hand, it probably does help the speed of
   some operations (such as main branch checkouts and updates).  */
#define CVSATTIC "Attic"

#define CVSLCK "#cvs.lock"
#define CVSRFL "#cvs.rfl"
#define CVSWFL "#cvs.wfl"
#define CVSRFLPAT "#cvs.rfl.*" /* wildcard expr to match read locks */
#define CVSEXT_LOG ",t"
#define CVSPREFIX ",,"
#define CVSDOTIGNORE ".cvsignore"
#define CVSDOTWRAPPER ".cvswrappers"

/* Command attributes - see function lookup_command_attribute(). */
#define CVS_CMD_IGNORE_ADMROOT 1

/* Set if CVS needs to create a CVS/Root file upon completion of this
   command.  The name may be slightly confusing, because the flag
   isn't really as general purpose as it seems (it is not set for cvs
   release).  */

#define CVS_CMD_USES_WORK_DIR 2

#define CVS_CMD_MODIFIES_REPOSITORY 4

/* miscellaneous CVS defines */

/* This is the string which is at the start of the non-log-message lines
   that we put up for the user when they edit the log message.  */
#define CVSEDITPREFIX "CVS: "
/* Number of characters in CVSEDITPREFIX to compare when deciding to strip
   off those lines.  We don't check for the space, to accomodate users who
   have editors which strip trailing spaces.  */
#define CVSEDITPREFIXLEN 4

#define CVSLCKAGE (60*60) /* 1-hour old lock files cleaned up */
#define CVSLCKSLEEP 30 /* wait 30 seconds before retrying */
#define CVSBRANCH "1.1.1" /* RCS branch used for vendor srcs */

#ifdef USE_VMS_FILENAMES
#define BAKPREFIX "_$"
#define DEVNULL "NLA0:"
#else /* USE_VMS_FILENAMES */
#define BAKPREFIX ".#" /* when rcsmerge'ing */
#ifndef DEVNULL
#define DEVNULL "/dev/null"
#endif
#endif /* USE_VMS_FILENAMES */

/*
 * Special tags. -rHEAD refers to the head of an RCS file, regardless of any
 * sticky tags. -rBASE refers to the current revision the user has checked
 * out This mimics the behaviour of RCS.
 */
#define TAG_HEAD "HEAD"
#define TAG_BASE "BASE"

/* Environment variable used by CVS */
#define CVSREAD_ENV "CVSREAD" /* make files read-only */
#define CVSREAD_DFLT 0 /* writable files by default */

#define TMPDIR_ENV "TMPDIR" /* Temporary directory */
/* #define TMPDIR_DFLT Set by options.h */
```

```
#define EDITOR1_ENV "CVSEDITOR" /* which editor to use */
#define EDITOR2_ENV "VISUAL" /* which editor to use */
#define EDITOR3_ENV "EDITOR" /* which editor to use */
/* #define    EDITOR_DFLT       Set by options.h */

#define CVSROOT_ENV "CVSROOT" /* source directory root */
#define CVSROOT_DFLT NULL /* No dflt; must set for checkout */

#define IGNORE_ENV "CVSIGNORE" /* More files to ignore */
#define WRAPPER_ENV     "CVSWRAPPERS"   /* name of the wrapper file */

#define CVSUMASK_ENV "CVSUMASK" /* Effective umask for repository */
/* #define    CVSUMASK_DFLT    Set by options.h */

/*
 * If the beginning of the Repository matches the following string, strip it
 * so that the output to the logfile does not contain a full pathname.
 *
 * If the CVSROOT environment variable is set, it overrides this define.
 */
#define REPOS_STRIP "/master/"

/* Large enough to hold DATEFORM.  Not an arbitrary limit as long as
   it is used for that purpose, and not to hold a string from the
   command line, the client, etc.  */
#define MAXDATELEN 50

/* The type of an entnode.  */
enum ent_type
{
   ENT_FILE, ENT_SUBDIR
};

/* structure of a entry record */
struct entnode
{
   enum ent_type type;
   char *user;
   char *version;

   /* Timestamp, or "" if none (never NULL).  */
   char *timestamp;

   /* Keyword expansion options, or "" if none (never NULL).  */
   char *options;

   char *tag;
   char *date;
   char *conflict;
   char *root;
   char *repository;
};
typedef struct entnode Entnode;

/* The type of request that is being done in do_module() */
enum mtype
{
   CHECKOUT, TAG, PATCH, EXPORT
};

/*
 * structure used for list-private storage by Entries_Open() and
 * Version_TS() and Find_Directories().
 */
struct stickydirtag
{
   /* These fields pass sticky tag information from Entries_Open() to
      Version_TS().  */
   int aflag;
   char *tag;
   char *date;
   int nonbranch;

   /* This field is set by Entries_Open() if there was subdirectory
      information; Find_Directories() uses it to see whether it needs
      to scan the directory itself.  */
   int subdirs;
};

/* Flags for find_{names,dirs} routines */
#define W_LOCAL      0x01   /* look for files locally */
#define W_REPOS      0x02   /* look for files in the repository */
#define W_ATTIC      0x04   /* look for files in the attic */

/* Flags for return values of direnter procs for the recursion processor */
enum direnter_type
{
   R_PROCESS = 1,          /* process files and maybe dirs */
   R_SKIP_FILES,           /* don't process files in this dir */
```

```
    R_SKIP_DIRS,                /* don't process sub-dirs */
    R_SKIP_ALL                  /* don't process files or dirs */
};
#ifdef ENUMS_CAN_BE_TROUBLE
typedef int Dtype;
#else
typedef enum direnter_type Dtype;
#endif

extern char *program_name, *program_path, *command_name;
extern char *Tmpdir, *Editor;
extern int cvsadmin_root;
extern char *CurDir;
extern int really_quiet, quiet;
extern int use_editor;
extern int cvswrite;
extern mode_t cvsumask;
extern int first_file_arg;
extern int handling_remotes;
extern int fetch_remotes;

/* Access method specified in CVSroot. */
typedef enum {
  local_method, server_method, pserver_method, kserver_method, gserver_method,
  ext_method
} CVSmethod;
extern char *method_names[]; /* change this in root.c if you change
                                the enum above */

extern char *CVSroot_original; /* the active, complete CVSroot string */
extern int client_active; /* nonzero if we are doing remote access */
extern CVSmethod CVSroot_method; /* one of the enum values above */
extern char *CVSroot_username; /* the username or NULL if method == local */
extern char *CVSroot_hostname; /* the hostname or NULL if method == local */
extern char *CVSroot_directory; /* the directory name */

extern char *emptydir_name PROTO ((void));

extern int trace;           /* Show all commands */
extern int noexec;          /* Don't modify disk anywhere */
extern int logoff;          /* Don't write history entry */

extern int top_level_admin;

#ifdef AUTH_SERVER_SUPPORT
extern char *Pserver_Repos;     /* used to check that same repos is
                                   transmitted in pserver auth and in
                                   CVS protocol. */
#endif /* AUTH_SERVER_SUPPORT */

extern char hostname[];

/* Externs that are included directly in the CVS sources */

int RCS_merge PROTO((RCSNode *, char *, char *, char *, char *, char *));
/* Flags used by RCS_* functions.  See the description of the individual
   functions for which flags mean what for each function.  */
#define RCS_FLAGS_FORCE 1
#define RCS_FLAGS_DEAD 2
#define RCS_FLAGS_QUIET 4
#define RCS_FLAGS_MODTIME 8
#define RCS_FLAGS_KEEPFILE 16

extern int RCS_exec_rcsdiff PROTO ((RCSNode *rcsfile,
                                    char *opts, char *options,
                                    char *rev1, char *rev2,
                                    char *label1, char *label2,
                                    char *workfile));
extern int diff_exec PROTO ((char *file1, char *file2, char *options,
                              char *out));
extern int diff_execv PROTO ((char *file1, char *file2,
                               char *label1, char *label2,
                               char *options, char *out));



#include "error.h"

DBM *open_module PROTO((void));
FILE *open_file PROTO((const char *, const char *));
List *Find_Directories PROTO((char *repository, int which, List *entries));
void Entries_Close PROTO((List *entries));
List *Entries_Open PROTO ((int aflag, char *update_dir));
void Subdirs_Known PROTO((List *entries));
void Subdir_Register PROTO((List *, const char *, const char *));
void Subdir_Deregister PROTO((List *, const char *, const char *));

char *Make_Date PROTO((char *rawdate));
char *date_from_time_t PROTO ((time_t));
```

```
char *Name_Repository PROTO((char *dir, char *update_dir));
char *Short_Repository PROTO((char *repository));
void Sanitize_Repository_Name PROTO((char *repository));

char *Name_Root PROTO((char *dir, char *update_dir));
int parse_cvsroot PROTO((char *CVSroot));
void set_local_cvsroot PROTO((char *dir));
void Create_Root PROTO((char *dir, char *rootdir));
void root_allow_add PROTO ((char *));
void root_allow_free PROTO ((void));
int root_allow_ok PROTO ((char *));

char *gca PROTO((const char *rev1, const char *rev2));
extern void check_numeric PROTO ((const char *, int, char **));
char *getcaller PROTO((void));
char *time_stamp PROTO((char *file));

char *xmalloc PROTO((size_t bytes));
void *xrealloc PROTO((void *ptr, size_t bytes));
void expand_string PROTO ((char **, size_t *, size_t));
char *xstrdup PROTO((const char *str));
void strip_trailing_newlines PROTO((char *str));
int pathname_levels PROTO ((char *path));

typedef int (*CALLPROC) PROTO((char *repository, char *value));
int Parse_Info PROTO((char *infofile, char *repository, CALLPROC callproc, int all));
extern int parse_config PROTO ((char *));

typedef RETSIGTYPE (*SIGCLEANUPPROC) PROTO(());
int SIG_register PROTO((int sig, SIGCLEANUPPROC sigcleanup));
int isdir PROTO((const char *file));
int isfile PROTO((const char *file));
int islink PROTO((const char *file));
int isdevice PROTO ((const char *));
int isreadable PROTO((const char *file));
int iswritable PROTO((const char *file));
int isaccessible PROTO((const char *file, const int mode));
int isabsolute PROTO((const char *filename));
char *xreadlink PROTO((const char *link));
char *last_component PROTO((char *path));
char *get_homedir PROTO ((void));
char *cvs_temp_name PROTO ((void));

int numdots PROTO((const char *s));
char *increment_revnum PROTO ((const char *));
int compare_revnums PROTO ((const char *, const char *));
int unlink_file PROTO((const char *f));
int unlink_file_dir PROTO((const char *f));
int update PROTO((int argc, char *argv[]));
int xcmp PROTO((const char *file1, const char *file2));
int yesno PROTO((void));
void *valloc PROTO((size_t bytes));
time_t get_date PROTO((char *date, struct timeb *now));
extern int Create_Admin PROTO ((char *dir, char *update_dir,
                                char *repository, char *tag, char *date,
                                int nonbranch, int warn));
extern int expand_at_signs PROTO ((char *, off_t, FILE *));

/* Locking subsystem (implemented in lock.c).  */

int Reader_Lock PROTO((char *xrepository));
void Lock_Cleanup PROTO((void));

/* Writelock an entire subtree, well the part specified by ARGC, ARGV, LOCAL,
   and AFLAG, anyway.  */
void lock_tree_for_write PROTO ((int argc, char **argv, int local, int aflag));

/* See lock.c for description.  */
extern void lock_dir_for_write PROTO ((char *));

void Scratch_Entry PROTO((List * list, char *fname));
void ParseTag PROTO((char **tagp, char **datep, int *nonbranchp));
void WriteTag PROTO ((char *dir, char *tag, char *date, int nonbranch,
                      char *update_dir, char *repository));
void cat_module PROTO((int status));
void check_entries PROTO((char *dir));
void close_module PROTO((DBM * db));
void copy_file PROTO((const char *from, const char *to));
void fperror PROTO((FILE * fp, int status, int errnum, char *message,...));
void free_names PROTO((int *pargc, char *argv[]));

extern int ign_name PROTO ((char *name));
void ign_add PROTO((char *ign, int hold));
void ign_add_file PROTO((char *file, int hold));
void ign_setup PROTO((void));
void ign_dir_add PROTO((char *name));
int ignore_directory PROTO((char *name));
typedef void (*Ignore_proc) PROTO ((char *, char *));
extern void ignore_files PROTO ((List *, List *, char *, Ignore_proc));
extern int ign_inhibit_server;
```

```
extern int ign_case;

#include "update.h"

void line2argv PROTO ((int *pargc, char ***argv, char *line, char *sepchars));
void make_directories PROTO((const char *name));
void make_directory PROTO((const char *name));
extern int mkdir_if_needed PROTO ((char *name));
void rename_file PROTO((const char *from, const char *to));
/* Expand wildcards in each element of (ARGC,ARGV).  This is according to the
   files which exist in the current directory, and accordingly to OS-specific
   conventions regarding wildcard syntax.  It might be desirable to change the
   former in the future (e.g. "cvs status *.h" including files which don't exist
   in the working directory).  The result is placed in *PARGC and *PARGV;
   the *PARGV array itself and all the strings it contains are newly
   malloc'd.  It is OK to call it with PARGC == &ARGC or PARGV == &ARGV.  */
extern void expand_wild PROTO ((int argc, char **argv,
                                int *pargc, char ***pargv));

#ifdef SERVER_SUPPORT
extern int cvs_casecmp PROTO ((char *, char *));
extern int fopen_case PROTO ((char *, char *, FILE **, char **));
#endif

void strip_trailing_slashes PROTO((char *path));
void update_delproc PROTO((Node * p));
void usage PROTO((const char *const *cpp));
void xchmod PROTO((char *fname, int writable));
char *xgetwd PROTO((void));
List *Find_Names PROTO((char *repository, int which, int aflag,
                        List ** optentries));
void Register PROTO((List * list, char *fname, char *vn, char *ts,
                     char *options, char *tag, char *date, char *ts_conflict, char* root, char *repository));
void Update_Logfile PROTO((char *repository, char *xmessage, FILE * xlogfp,
                           List * xchanges));
void do_editor PROTO((char *dir, char **messagep,
                      char *repository, List * changes));

void do_verify PROTO((char *message, char *repository));

typedef int (*CALLBACKPROC) PROTO((int *pargc, char *argv[], char *where,
        char *mwhere, char *mfile, int shorten, int local_specified,
        char *omodule, char *msg));

/* This is the structure that the recursion processor passes to the
   fileproc to tell it about a particular file.  */
struct file_info
{
    /* Name of the file, without any directory component.  */
    char *file;

    /* Name of the directory we are in, relative to the directory in
       which this command was issued.  We have cd'd to this directory
       (either in the working directory or in the repository, depending
       on which sort of recursion we are doing).  If we are in the directory
       in which the command was issued, this is "".  */
    char *update_dir;

    /* update_dir and file put together, with a slash between them as
       necessary.  This is the proper way to refer to the file in user
       messages.  */
    char *fullname;

    /* Name of the directory corresponding to the repository which contains
       this file.  */
    char *repository;

    /* The pre-parsed entries for this directory.  */
    List *entries;

    RCSNode *rcs;
};

typedef int (*FILEPROC) PROTO ((void *callerdat, struct file_info *finfo));
typedef int (*FILESDONEPROC) PROTO ((void *callerdat, int err,
                                     char *repository, char *update_dir,
                                     List *entries));
typedef Dtype (*DIRENTPROC) PROTO ((void *callerdat, char *dir,
                                    char *repos, char *update_dir,
                                    List *entries));
typedef int (*DIRLEAVEPROC) PROTO ((void *callerdat, char *dir, int err,
                                    char *update_dir, List *entries));

extern int mkmodules PROTO ((char *dir));
extern int init PROTO ((int argc, char **argv));

int do_module PROTO((DBM * db, char *mname, enum mtype m_type, char *msg,
                     CALLBACKPROC callback_proc, char *where, int shorten,
                     int local_specified, int run_module_prog, char *extra_arg));
void history_write PROTO((int type, char *update_dir, char *revs, char *name,
```

```
                    char *repository));
int start_recursion PROTO((FILEPROC fileproc, FILESDONEPROC filesdoneproc,
                    DIRENTPROC direntproc, DIRLEAVEPROC dirleaveproc,
                    void *callerdat,
                    int argc, char *argv[], int local, int which,
                    int aflag, int readlock, char *update_preload,
                    int dosrcs));
void SIG_beginCrSect PROTO((void));
void SIG_endCrSect PROTO((void));
void read_cvsrc PROTO((int *argc, char ***argv, char *cmdname));

char *make_message_rcslegal PROTO((char *message));
extern int file_has_markers PROTO ((const struct file_info *));
extern void get_file PROTO ((const char *, const char *, const char *,
                    char **, size_t *, size_t *));

/* flags for run_exec(), the fast system() for CVS */
#define RUN_NORMAL 0x0000 /* no special behaviour */
#define RUN_COMBINED 0x0001 /* stdout is duped to stderr */
#define RUN_REALLY  0x0002 /* do the exec, even if noexec is on */
#define RUN_STDOUT_APPEND 0x0004 /* append to stdout, don't truncate */
#define RUN_STDERR_APPEND 0x0008 /* append to stderr, don't truncate */
#define RUN_SIGIGNORE 0x0010 /* ignore interrupts for command */
#define RUN_TTY (char *)0   /* for the benefit of lint */

void run_arg PROTO((const char *s));
void run_print PROTO((FILE * fp));
void run_setup PROTO ((const char *prog));
int run_exec PROTO((const char *stin, const char *stout, const char *sterr,
                    int flags));

/* other similar-minded stuff from run.c.  */
FILE *run_popen PROTO((const char *, const char *));
int piped_child PROTO((char **, int *, int *));
void close_on_exec PROTO((int));
int filter_stream_through_program PROTO((int, int, char **, pid_t *));

pid_t waitpid PROTO((pid_t, int *, int));

/*
 * a struct vers_ts contains all the information about a file including the
 * user and rcs file names, and the version checked out and the head.
 *
 * this is usually obtained from a call to Version_TS which takes a
 * tag argument for the RCS file if desired
 */
struct vers_ts
{
    /* rcs version user file derives from, from CVS/Entries.
       It can have the following special values:

       NULL = file is not mentioned in Entries (this is also used for a
              directory).
       "" = ILLEGAL!  The comment used to say that it meant "no user file"
            but as far as I know CVS didn't actually use it that way.
            Note that according to cvs.texinfo, "" is not legal in the
            Entries file.
       0 = user file is new
       -vers = user file to be removed.  */
    char *vn_user;

    /* Numeric revision number corresponding to ->vn_tag (->vn_tag
       will often be symbolic).  */
    char *vn_rcs;
    /* If ->tag is a simple tag in the RCS file--a tag which really
       exists which is not a magic revision--and if ->date is NULL,
       then this is a copy of ->tag.  Otherwise, it is a copy of
       ->vn_rcs.  */
    char *vn_tag;
   /* -> vn_remote is the remote tag, in the server:tag form */
    char *vn_remote;

    /* This is the timestamp from stating the file in the working directory.
       It is NULL if there is no file in the working directory.  It is
       "Is-modified" if we know the file is modified but don't have its
       contents.  */
    char *ts_user;
    /* Timestamp from CVS/Entries.  For the server, ts_user and ts_rcs
       are computed in a slightly different way, but the fact remains that
       if they are equal the file in the working directory is unmodified
       and if they differ it is modified.  */
    char *ts_rcs;

    /* Options from CVS/Entries (keyword expansion), malloc'd.  If none,
       then it is an empty string (never NULL).  */
    char *options;

    /* If non-NULL, there was a conflict (or merely a merge?  See merge_file)
       and the time stamp in this field is the time stamp of the working
       directory file which was created with the conflict markers in it.
```

```
         This is from CVS/Entries.  */
720      char *ts_conflict;

         /* Tag specified on the command line, or if none, tag stored in
            CVS/Entries.  */
         char *tag;
         /* Date specified on the command line, or if none, date stored in
            CVS/Entries.  */
         char *date;
         /* If this is 1, then tag is not a branch tag.  If this is 0, then
            tag may or may not be a branch tag.  */
730      int nonbranch;

         /* Pointer to entries file node  */
         Entnode *entdata;

         /* Pointer to parsed src file info */
         RCSNode *srcfile;
       };
       typedef struct vers_ts Vers_TS;

740    Vers_TS *Version_TS PROTO ((struct file_info *finfo, char *options, char *tag,
                                   char *date, int force_tag_match,
                                   int set_time));
       void freevers_ts PROTO ((Vers_TS ** versp));

       /* Miscellaneous CVS infrastructure which layers on top of the recursion
          processor (for example, needs struct file_info).  */

       int Checkin PROTO ((int type, struct file_info *finfo, char *rcs, char *rev,
                           char *tag, char *options, char *message));
750    int No_Difference PROTO ((struct file_info *finfo, Vers_TS *vers));
       /* TODO: can the finfo argument to special_file_mismatch be changed? -twp */
       int special_file_mismatch PROTO ((struct file_info *finfo,
                                         char *rev1, char *rev2));

       /* CVSADM_BASEREV stuff, from entries.c.  */
       extern char *base_get PROTO ((struct file_info *));
       extern void base_register PROTO ((struct file_info *, char *));
       extern void base_deregister PROTO ((struct file_info *));

760    /*
        * defines for Classify_File() to determine the current state of a file.
        * These are also used as types in the data field for the list we make for
        * Update_Logfile in commit, import, and add.
        */
       enum classify_type
       {
         T_UNKNOWN = 1,          /* no old-style analog existed  */
         T_CONFLICT,             /* C (conflict) list            */
         T_NEEDS_MERGE,          /* G (needs merging) list       */
770      T_MODIFIED,             /* M (needs checked in) list    */
         T_CHECKOUT,             /* O (needs checkout) list      */
         T_ADDED,                /* A (added file) list          */
         T_REMOVED,              /* R (removed file) list        */
         T_REMOVE_ENTRY,         /* W (removed entry) list       */
         T_UPTODATE,             /* File is up-to-date           */
       #ifdef SERVER_SUPPORT
         T_PATCH,                /* P Like C, but can patch      */
       #endif
         T_TITLE,                /* title for node type          */
780      T_REMOTE                        /* go to another server and ask  */
       };
       typedef enum classify_type Ctype;

       Ctype Classify_File PROTO
         ((struct file_info *finfo, char *tag, char *date, char *options,
           int force_tag_match, int aflag, Vers_TS **versp, int pipeout));

       /*
        * structure used for list nodes passed to Update_Logfile() and
790     * do_editor().
        */
       struct logfile_info
       {
        enum classify_type type;
        char *tag;
        char *rev_old;          /* rev number before a commit/modify,
                                   NULL for add or import */
        char *rev_new;          /* rev number after a commit/modify,
                                   add, or import, NULL for remove */
800    };

       /* Wrappers.  */

       typedef enum { WRAP_MERGE, WRAP_COPY } WrapMergeMethod;
       typedef enum {
         /* -t and -f wrapper options.  Treating directories as single files.  */
         WRAP_TOCVS,
         WRAP_FROMCVS,
```

```
    /* -k wrapper option.  Default keyword expansion options.  */
    WRAP_RCSOPTION
} WrapMergeHas;

void  wrap_setup PROTO((void));
int   wrap_name_has PROTO((const char *name,WrapMergeHas has));
char *wrap_rcsoption PROTO ((const char *fileName, int asFlag));
char *wrap_tocvs_process_file PROTO((const char *fileName));
int   wrap_merge_is_copy PROTO((const char *fileName));
void wrap_fromcvs_process_file PROTO ((const char *fileName));
void wrap_add_file PROTO((const char *file,int temp));
void wrap_add PROTO((char *line,int temp));
void wrap_send PROTO ((void));
#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
void wrap_unparse_rcs_options PROTO ((char **, int));
#endif /* SERVER_SUPPORT || CLIENT_SUPPORT */

/* Pathname expansion */
char *expand_path PROTO((char *name, char *file, int line));

/* User variables.  */
extern List *variable_list;

extern void variable_set PROTO ((char *nameval));

int watch PROTO ((int argc, char **argv));
int edit PROTO ((int argc, char **argv));
int unedit PROTO ((int argc, char **argv));
int editors PROTO ((int argc, char **argv));
int watchers PROTO ((int argc, char **argv));
extern int annotate PROTO ((int argc, char **argv));
extern int add PROTO ((int argc, char **argv));
extern int admin PROTO ((int argc, char **argv));
extern int checkout PROTO ((int argc, char **argv));
extern int commit PROTO ((int argc, char **argv));
extern int diff PROTO ((int argc, char **argv));
extern int history PROTO ((int argc, char **argv));
extern int import PROTO ((int argc, char **argv));
extern int cvslog PROTO ((int argc, char **argv));
#ifdef AUTH_CLIENT_SUPPORT
extern int login PROTO((int argc, char **argv));
int logout PROTO((int argc, char **argv));
#endif /* AUTH_CLIENT_SUPPORT */
extern int patch PROTO((int argc, char **argv));
extern int release PROTO((int argc, char **argv));
extern int cvsremove PROTO((int argc, char **argv));
extern int rtag PROTO((int argc, char **argv));
extern int cvsstatus PROTO((int argc, char **argv));
extern int cvstag PROTO((int argc, char **argv));

extern unsigned long int lookup_command_attribute PROTO((char *));

#if defined(AUTH_CLIENT_SUPPORT) || defined(AUTH_SERVER_SUPPORT)
char *scramble PROTO ((char *str));
char *descramble PROTO ((char *str));
#endif /* AUTH_CLIENT_SUPPORT || AUTH_SERVER_SUPPORT */

#define AUTH_GSSAPI 1
#define AUTH_KERBEROS_V4 2
#define AUTH_PASSWORD 3

#ifdef AUTH_CLIENT_SUPPORT
char *get_cvs_password PROTO((void));
#endif /* AUTH_CLIENT_SUPPORT */

extern void tag_check_valid PROTO ((char *, int, char **, int, int, char *));
extern void tag_check_valid_join PROTO ((char *, int, char **, int, int,
                                         char *));

/* From server.c and documented there.  */
extern void cvs_output PROTO ((const char *, size_t));
extern void cvs_output_binary PROTO ((char *, size_t));
extern void cvs_outerr PROTO ((const char *, size_t));
extern void cvs_flusherr PROTO ((void));
extern void cvs_flushout PROTO ((void));
extern void cvs_output_tagged PROTO ((char *, char *));

#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
#include "server.h"
#endif
```

A.13 cvsrc.c

```
/*
 * Copyright (c) 1993 david d zuhn
 *
 * Written by david d 'zoo' zuhn while at Cygnus Support
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 */

#include "cvs.h"
#include "getline.h"

/* this file is to be found in the user's home directory */

#ifndef CVSRC_FILENAME
#define CVSRC_FILENAME ".cvsrc"
#endif
char cvsrc[] = CVSRC_FILENAME;

#define GROW 10

extern char *strtok ();

/* Read cvsrc, processing options matching CMDNAME ("cvs" for global
   options, and update *ARGC and *ARGV accordingly.  */

void
read_cvsrc (argc, argv, cmdname)
    int *argc;
    char ***argv;
    char *cmdname;
{
    char *homedir;
    char *homeinit;
    FILE *cvsrcfile;

    char *line;
    int line_length;
    size_t line_chars_allocated;

    char *optstart;

    int command_len;
    int found = 0;

    int i;

    int new_argc;
    int max_new_argv;
    char **new_argv;

    /* old_argc and old_argv hold the values returned from the
       previous invocation of read_cvsrc and are used to free the
       allocated memory.  The first invocation of read_cvsrc gets argv
       from the system, this memory must not be free'd.  */
    static int old_argc = 0;
    static char **old_argv = NULL;

    /* don't do anything if argc is -1, since that implies "help" mode */
    if (*argc == -1)
        return;

    /* determine filename for ~/.cvsrc */

    homedir = get_homedir ();
    if (!homedir)
        return;

    homeinit = (char *) xmalloc (strlen (homedir) + strlen (cvsrc) + 10);
    strcpy (homeinit, homedir);
    strcat (homeinit, "/");
    strcat (homeinit, cvsrc);

    /* if it can't be read, there's no point to continuing */

    if (!isreadable (homeinit))
    {
        free (homeinit);
        return;
    }

    /* now scan the file until we find the line for the command in question */

    line = NULL;
    line_chars_allocated = 0;
    command_len = strlen (cmdname);
```

```
    cvsrcfile = open_file (homeinit, "r");
    while ((line_length = getline (&line, &line_chars_allocated, cvsrcfile))
           >= 0)
    {
        /* skip over comment lines */
        if (line[0] == '#')
            continue;

        /* stop if we match the current command */
        if (!strncmp (line, cmdname, command_len)
            && isspace (*(line + command_len)))
        {
            found = 1;
            break;
        }
    }

    if (line_length < 0 && !feof (cvsrcfile))
        error (0, errno, "cannot read %s", homeinit);

    fclose (cvsrcfile);

    /* setup the new options list */

    new_argc = 1;
    max_new_argv = (*argc) + GROW;
    new_argv = (char **) xmalloc (max_new_argv * sizeof (char*));
    new_argv[0] = xstrdup ((*argv)[0]);

    if (found)
    {
        /* skip over command in the options line */
        for (optstart = strtok (line + command_len, "\t \n");
             optstart;
             optstart = strtok (NULL, "\t \n"))
        {
            new_argv [new_argc++] = xstrdup (optstart);

            if (new_argc >= max_new_argv)
            {
                max_new_argv += GROW;
                new_argv = (char **) xrealloc (new_argv, max_new_argv * sizeof (char*));
            }
        }
    }

    if (line != NULL)
        free (line);

    /* now copy the remaining arguments */

    if (new_argc + *argc > max_new_argv)
    {
        max_new_argv = new_argc + *argc;
        new_argv = (char **) xrealloc (new_argv, max_new_argv * sizeof (char*));
    }
    for (i=1; i < *argc; i++)
    {
        new_argv [new_argc++] = xstrdup ((*argv)[i]);
    }

    if (old_argv != NULL)
    {
        /* Free the memory which was allocated in the previous
           read_cvsrc call.  */
        free_names (&old_argc, old_argv);
    }

    old_argc = *argc = new_argc;
    old_argv = *argv = new_argv;

    free (homeinit);
    return;
}
```

A.14 diff.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Difference
 *
 * Run diff against versions in the repository.  Options that are specified are
 * passed on directly to "rcsdiff".
 *
 * Without any file arguments, runs diff against all the currently modified
 * files.
 */

#include "cvs.h"

enum diff_file
{
    DIFF_ERROR,
    DIFF_ADDED,
    DIFF_REMOVED,
    DIFF_DIFFERENT,
    DIFF_SAME
};

static Dtype diff_dirproc PROTO ((void *callerdat, char *dir,
                                  char *pos_repos, char *update_dir,
                                  List *entries));
static int diff_filesdoneproc PROTO ((void *callerdat, int err,
                                      char *repos, char *update_dir,
                                      List *entries));
static int diff_dirleaveproc PROTO ((void *callerdat, char *dir,
                                     int err, char *update_dir,
                                     List *entries));
static enum diff_file diff_file_nodiff PROTO ((struct file_info *finfo,
                                               Vers_TS *vers,
                                               enum diff_file));
static int diff_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static void diff_mark_errors PROTO((int err));

static char *diff_rev1, *diff_rev2;
static char *diff_date1, *diff_date2;
static char *use_rev1, *use_rev2;
static int have_rev1_label, have_rev2_label;

/* Revision of the user file, if it is unchanged from something in the
   repository and we want to use that fact.  */
static char *user_file_rev;

static char *options;
static char *opts;
static size_t opts_allocated = 1;
static int diff_errors;
static int empty_files = 0;

/* FIXME: should be documenting all the options here.  They don't
   perfectly match rcsdiff options (for example, we always support
   -ifdef and -context, but rcsdiff only does if diff does).  */
static const char *const diff_usage[] =
{
    "Usage: %s %s [-lNR] [rcsdiff-options]\n",
    "    [[-r rev1 | -D date1] [-r rev2 | -D date2]] [files. . .] \n",
    "\t-l\tLocal directory only, not recursive\n",
    "\t-R\tProcess directories recursively.\n",
    "\t-D d1\tDiff revision for date against working file.\n",
    "\t-D d2\tDiff rev1/date1 against date2.\n",
    "\t-N\tinclude diffs for added and removed files.\n",
    "\t-r rev1\tDiff revision for rev1 against working file.\n",
    "\t-r rev2\tDiff rev1/date1 against rev2.\n",
    "\t--ifdef=arg\tOutput diffs in ifdef format.\n",
    "(consult the documentation for your diff program for rcsdiff-options.\n",
    "The most popular is -c for context diffs but there are many more).\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

/* I copied this array directly out of diff.c in diffutils 2.7, after
   removing the following entries, none of which seem relevant to use
   with CVS:
     -help
     -version
     -recursive
     -unidirectional-new-file
     -starting-file
     -exclude
     -exclude-from
```

```
      –sdiff-merge-assist
90
    I changed the options which take optional arguments (–context and
    –unified) to return a number rather than a letter, so that the
    optional argument could be handled more easily.  I changed the
    –paginate and –brief options to return a number, since -l and -q
    mean something else to cvs diff.

    The numbers 129- that appear in the fourth element of some entries
    tell the big switch in 'diff' how to process those options. – Ian

100 The following options, which diff lists as "An alias, no longer
    recommended" have been removed: –file-label –entire-new-file
    –ascii –print.  */

  static struct option const longopts[] =
  {
    {"ignore-blank-lines", 0, 0, 'B'},
    {"context", 2, 0, 143},
    {"ifdef", 1, 0, 131},
    {"show-function-line", 1, 0, 'F'},
110 {"speed-large-files", 0, 0, 'H'},
    {"ignore-matching-lines", 1, 0, 'I'},
    {"label", 1, 0, 'L'},
    {"new-file", 0, 0, 'N'},
    {"initial-tab", 0, 0, 148},
    {"width", 1, 0, 'W'},
    {"text", 0, 0, 'a'},
    {"ignore-space-change", 0, 0, 'b'},
    {"minimal", 0, 0, 'd'},
    {"ed", 0, 0, 'e'},
120 {"forward-ed", 0, 0, 'f'},
    {"ignore-case", 0, 0, 'i'},
    {"paginate", 0, 0, 144},
    {"rcs", 0, 0, 'n'},
    {"show-c-function", 0, 0, 'p'},

    /* This is a potentially very useful option, except the output is so
       silly.  It would be much better for it to look like "cvs rdiff -s"
       which displays all the same info, minus quite a few lines of
       extraneous garbage.  */
130 {"brief", 0, 0, 145},

    {"report-identical-files", 0, 0, 's'},
    {"expand-tabs", 0, 0, 't'},
    {"ignore-all-space", 0, 0, 'w'},
    {"side-by-side", 0, 0, 147},
    {"unified", 2, 0, 146},
    {"left-column", 0, 0, 129},
    {"suppress-common-lines", 0, 0, 130},
    {"old-line-format", 1, 0, 132},
140 {"new-line-format", 1, 0, 133},
    {"unchanged-line-format", 1, 0, 134},
    {"line-format", 1, 0, 135},
    {"old-group-format", 1, 0, 136},
    {"new-group-format", 1, 0, 137},
    {"unchanged-group-format", 1, 0, 138},
    {"changed-group-format", 1, 0, 139},
    {"horizon-lines", 1, 0, 140},
    {"binary", 0, 0, 142},
    {0, 0, 0, 0}
150 };

  /* CVS 1.9 and similar versions seemed to have pretty weird handling
     of -y and -T.  In the cases where it called rcsdiff,
     they would have the meanings mentioned below.  In the cases where it
     called diff, they would have the meanings mentioned in "longopts".
     Noone seems to have missed them, so I think the right thing to do is
     just to remove the options altogether (which I have done).

     In the case of -z and -q, "cvs diff" did not accept them even back
160  when we called rcsdiff (at least, it hasn't accepted them
     recently).

     In comparing rcsdiff to the new CVS implementation, I noticed that
     the following rcsdiff flags are not handled by CVS diff:

          -y: perform diff even when the requested revisions are the
              same revision number
          -q: run quietly
          -T: preserve modification time on the RCS file
170       -z: specify timezone for use in file labels

     I think these are not really relevant.  -y is undocumented even in
     RCS 5.7, and seems like a minor change at best.  According to RCS
     documentation, -T only applies when a RCS file has been modified
     because of lock changes; doesn't CVS sidestep RCS's entire lock
     structure?  -z seems to be unsupported by CVS diff, and has a
     different meaning as a global option anyway.  (Adding it could be
     a feature, but if it is left out for now, it should not break
```

```
   anything.)  For the purposes of producing output, CVS diff appears
   mostly to ignore -q.  Maybe this should be fixed, but I think it's
   a larger issue than the changes included here.  */

static void strcat_and_allocate PROTO ((char **, size_t *, const char *));

/* *STR is a pointer to a malloc'd string.  *LENP is its allocated
   length.  Add SRC to the end of it, reallocating if necessary.  */
static void
strcat_and_allocate (str, lenp, src)
    char **str;
    size_t *lenp;
    const char *src;
{
    size_t new_size;

    new_size = strlen (*str) + strlen (src) + 1;
    if (*str == NULL || new_size >= *lenp)
    {
        while (new_size >= *lenp)
            *lenp *= 2;
        *str = xrealloc (*str, *lenp);
    }
    strcat (*str, src);
}

int
diff (argc, argv)
    int argc;
    char **argv;
{
    char tmp[50];
    int c, err = 0;
    int local = 0;
    int which;
    int option_index;

    if (argc == -1)
        usage (diff_usage);

    have_rev1_label = have_rev2_label = 0;

    /*
     * Note that we catch all the valid arguments here, so that we can
     * intercept the -r arguments for doing revision diffs; and -l/-R for a
     * non-recursive/recursive diff.
     */

    /* For server, need to be able to do this command more than once
       (according to the protocol spec, even if the current client
       doesn't use it).  */
    if (opts == NULL)
    {
        opts_allocated = 1;
        opts = xmalloc (opts_allocated);
    }
    opts[0] = '\0';

    optind = 0;
    while ((c = getopt_long (argc, argv,
            "+abcdefhilnpstuw0123456789BHNRC:D:F:I:L:U:V:W:k:r:",
                        longopts, &option_index)) != -1)
    {
        switch (c)
        {
            case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
            case 'h': case 'i': case 'n': case 'p': case 's': case 't':
            case 'u': case 'w': case '0': case '1': case '2':
            case '3': case '4': case '5': case '6': case '7': case '8':
            case '9': case 'B': case 'H':
                (void) sprintf (tmp, " -%c", (char) c);
                strcat_and_allocate (&opts, &opts_allocated, tmp);
                break;
            case 'L':
                if (have_rev1_label++)
                    if (have_rev2_label++)
                    {
                        error (0, 0, "extra -L arguments ignored");
                        break;
                    }

                strcat_and_allocate (&opts, &opts_allocated, " -L");
                strcat_and_allocate (&opts, &opts_allocated, optarg);
                break;
            case 'C': case 'F': case 'I': case 'U': case 'V': case 'W':
                (void) sprintf (tmp, " -%c", (char) c);
                strcat_and_allocate (&opts, &opts_allocated, tmp);
                strcat_and_allocate (&opts, &opts_allocated, optarg);
                break;
            case 131:
```

```
                /* -ifdef.  */
                strcat_and_allocate (&opts, &opts_allocated, " -D");
                strcat_and_allocate (&opts, &opts_allocated, optarg);
                break;
            case 129: case 130:             case 132: case 133: case 134:
            case 135: case 136: case 137: case 138: case 139: case 140:
            case 141: case 142: case 143: case 144: case 145: case 146:
            case 147: case 148:
                strcat_and_allocate (&opts, &opts_allocated, " --");
                strcat_and_allocate (&opts, &opts_allocated,
                                     longopts[option_index].name);
                if (longopts[option_index].has_arg == 1
                    || (longopts[option_index].has_arg == 2
                        && optarg != NULL))
                {
                    strcat_and_allocate (&opts, &opts_allocated, "=");
                    strcat_and_allocate (&opts, &opts_allocated, optarg);
                }
                break;
            case 'R':
                local = 0;
                break;
            case 'l':
                local = 1;
                break;
            case 'k':
                if (options)
                    free (options);
                options = RCS_check_kflag (optarg);
                break;
            case 'r':
                if (diff_rev2 != NULL || diff_date2 != NULL)
                    error (1, 0,
                       "no more than two revisions/dates can be specified");
                if (diff_rev1 != NULL || diff_date1 != NULL)
                    diff_rev2 = optarg;
                else
                    diff_rev1 = optarg;
                break;
            case 'D':
                if (diff_rev2 != NULL || diff_date2 != NULL)
                    error (1, 0,
                       "no more than two revisions/dates can be specified");
                if (diff_rev1 != NULL || diff_date1 != NULL)
                    diff_date2 = Make_Date (optarg);
                else
                    diff_date1 = Make_Date (optarg);
                break;
            case 'N':
                empty_files = 1;
                break;
            case '?':
            default:
                usage (diff_usage);
                break;
        }
    }
    if (client_active && !handling_remotes)
      first_file_arg = argc;
    argc -= optind;
    argv += optind;

    /* make sure options is non-null */
    if (!options)
        options = xstrdup ("");

#ifdef CLIENT_SUPPORT
    if (client_active) {
      fetch_remotes = 1;
        /* We're the client side.  Fire up the remote server.  */
        start_server ();

        ign_setup ();

        if (local)
            send_arg("-l");
        if (empty_files)
            send_arg("-N");
        send_option_string (opts);
        if (options[0] != '\0')
            send_arg (options);
        if (diff_rev1)
            option_with_arg ("-r", diff_rev1);
        if (diff_date1)
            client_senddate (diff_date1);
        if (diff_rev2)
            option_with_arg ("-r", diff_rev2);
        if (diff_date2)
            client_senddate (diff_date2);
```

```
360         send_file_names (argc, argv, SEND_EXPAND_WILD);

            /* Send the current files unless diffing two revs from the archive */
            if (diff_rev2 == NULL && diff_date2 == NULL)
                send_files (argc, argv, local, 0, 0);
            else
                send_files (argc, argv, local, 0, SEND_NO_CONTENTS);

            send_to_server ("diff\012", 0);
            err = get_responses_and_close ();
370         if (handling_remotes)
              return 2;

            if (options != NULL) {
            free (options);
              options = NULL;
            }
            if (diff_rev1 != NULL) {
              free (diff_rev1);
              diff_rev1 = NULL;
380         }
            if (diff_rev2 != NULL) {
              free (diff_rev2);
              diff_rev2 = NULL;
            }
            if (diff_date1 != NULL) {
              free (diff_date1);
              diff_date1 = NULL;
            }
            if (diff_date2 != NULL) {
390           free (diff_date2);
              diff_date2 = NULL;
            }
            return (err);
        }
    #endif

        if (diff_rev1 != NULL)
            tag_check_valid (diff_rev1, argc, argv, local, 0, "");
        if (diff_rev2 != NULL)
400         tag_check_valid (diff_rev2, argc, argv, local, 0, "");

        which = W_LOCAL;
        if (diff_rev1 != NULL || diff_date1 != NULL)
            which |= W_REPOS | W_ATTIC;

        wrap_setup ();

        /* start the recursion processor */
        err = start_recursion (diff_fileproc, diff_filesdoneproc, diff_dirproc,
410                            diff_dirleaveproc, NULL, argc, argv, local,
                               which, 0, 1, (char *) NULL, 1);

        /* clean up */
        free (options);
        return (err);
    }

    /*
     * Do a file diff
420  */
    /* ARGSUSED */
    static int
    diff_fileproc (callerdat, finfo)
        void *callerdat;
        struct file_info *finfo;
    {
        int status, err = 2;            /* 2 == trouble, like rcsdiff */
        Vers_TS *vers;
        enum diff_file empty_file = DIFF_DIFFERENT;
430     char *tmp;
        char *tocvsPath;
        char *fname;

        /* Initialize these solely to avoid warnings from gcc -Wall about
           variables that might be used uninitialized.  */
        tmp = NULL;
        fname = NULL;

        user_file_rev = 0;
440     vers = Version_TS (finfo, NULL, NULL, NULL, 1, 0);

    #ifdef SERVER_SUPPORT
        {
          int have_remote = 0;

          Vers_TS* vers1 = Version_TS (finfo, NULL, diff_rev1, diff_date1, 1, 0);
          Vers_TS* vers2 = Version_TS (finfo, NULL, diff_rev2, diff_date2, 1, 0);
```

```
    if ((vers1 != NULL) && (vers1 -> vn_remote != NULL)) {
      server_output_not_carried_for_file (finfo, vers1);
      have_remote = 1;
    }

    if ((vers2 != NULL) && (vers2 -> vn_remote != NULL)) {
      server_output_not_carried_for_file (finfo, vers2);
      have_remote = 1;
    }

    if (have_remote)
      return 0;
  }
#endif

  if (diff_rev2 != NULL || diff_date2 != NULL)
  {
    /* Skip all the following checks regarding the user file; we're
       not using it. */
  }
  else if (vers->vn_user == NULL)
  {
    /* The file does not exist in the working directory. */
    if ((diff_rev1 != NULL || diff_date1 != NULL)
        && vers->srcfile != NULL)
    {
      /* The file does exist in the repository. */
      if (empty_files)
        empty_file = DIFF_REMOVED;
      else
      {
        int exists;

        exists = 0;
        /* special handling for TAG_HEAD */
        if (diff_rev1 && strcmp (diff_rev1, TAG_HEAD) == 0)
        {
          char *head =
            (vers->vn_rcs == NULL
             ? NULL
             : RCS_branch_head (vers->srcfile, vers->vn_rcs));
          exists = head != NULL;
          if (head != NULL)
            free (head);
        }
        else
        {
          Vers_TS *xvers;

          xvers = Version_TS (finfo, NULL, diff_rev1, diff_date1,
                              1, 0);
          exists = xvers->vn_rcs != NULL;
          freevers_ts (&xvers);
        }
        if (exists)
          error (0, 0,
                 "%s no longer exists, no comparison available",
                 finfo->fullname);
        freevers_ts (&vers);
        diff_mark_errors (err);
        return (err);
      }
    }
    else
    {
      error (0, 0, "I know nothing about %s", finfo->fullname);
      freevers_ts (&vers);
      diff_mark_errors (err);
      return (err);
    }
  }
  else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '\0')
  {
    if (empty_files)
      empty_file = DIFF_ADDED;
    else
    {
      error (0, 0, "%s is a new entry, no comparison available",
             finfo->fullname);
      freevers_ts (&vers);
      diff_mark_errors (err);
      return (err);
    }
  }
  else if (vers->vn_user[0] == '-')
  {
    if (empty_files)
      empty_file = DIFF_REMOVED;
    else
    {
```

```
            error (0, 0, "%s was removed, no comparison available",
                   finfo->fullname);
            freevers_ts (&vers);
            diff_mark_errors (err);
            return (err);
        }
    }
    else
    {
        if (vers->vn_rcs == NULL && vers->srcfile == NULL)
        {
            error (0, 0, "cannot find revision control file for %s",
                   finfo->fullname);
            freevers_ts (&vers);
            diff_mark_errors (err);
            return (err);
        }
        else
        {
            if (vers->ts_user == NULL)
            {
                error (0, 0, "cannot find %s", finfo->fullname);
                freevers_ts (&vers);
                diff_mark_errors (err);
                return (err);
            }
            else if (!strcmp (vers->ts_user, vers->ts_rcs))
            {
                /* The user file matches some revision in the repository
                   Diff against the repository (for remote CVS, we might not
                   have a copy of the user file around).  */
                user_file_rev = vers->vn_user;
            }
        }
    }

    empty_file = diff_file_nodiff (finfo, vers, empty_file);
    if (empty_file == DIFF_SAME || empty_file == DIFF_ERROR)
    {
        freevers_ts (&vers);
        if (empty_file == DIFF_SAME)
        {
            /* In the server case, would be nice to send a "Checked-in"
               response, so that the client can rewrite its timestamp.
               server_checked_in by itself isn't the right thing (it
               needs a server_register), but I'm not sure what is.
               It isn't clear to me how "cvs status" handles this (that
               is, for a client which sends Modified not Is-modified to
               "cvs status"), but it does.  */
            return (0);
        }
        else
        {
            diff_mark_errors (err);
            return (err);
        }
    }

    if (empty_file == DIFF_DIFFERENT)
    {
        int dead1, dead2;

        if (use_rev1 == NULL)
            dead1 = 0;
        else
            dead1 = RCS_isdead (vers->srcfile, use_rev1);
        if (use_rev2 == NULL)
            dead2 = 0;
        else
            dead2 = RCS_isdead (vers->srcfile, use_rev2);

        if (dead1 && dead2)
        {
            freevers_ts (&vers);
            return (0);
        }
        else if (dead1)
        {
            if (empty_files)
                empty_file = DIFF_ADDED;
            else
            {
                error (0, 0, "%s is a new entry, no comparison available",
                       finfo->fullname);
                freevers_ts (&vers);
                diff_mark_errors (err);
                return (err);
            }
        }
        else if (dead2)
```

```
        {
630         if (empty_files)
                empty_file = DIFF_REMOVED;
            else
            {
                error (0, 0, "%s was removed, no comparison available",
                       finfo->fullname);
                freevers_ts (&vers);
                diff_mark_errors (err);
                return (err);
            }
640     }
    }

    /* Output an "Index:" line for patch to use */
    cvs_output ("Index: ", 0);
    cvs_output (finfo->fullname, 0);
    cvs_output ("\n", 1);

    tocvsPath = wrap_tocvs_process_file(finfo->file);
    if (tocvsPath)
650 {
        /* Backup the current version of the file to CVS/,,filename */
        fname = xmalloc (strlen (finfo->file)
                         + sizeof CVSADM
                         + sizeof CVSPREFIX
                         + 10);
        sprintf(fname,"%s/%s%s",CVSADM, CVSPREFIX, finfo->file);
        if (unlink_file_dir (fname) < 0)
            if (! existence_error (errno))
                error (1, errno, "cannot remove %s", fname);
660     rename_file (finfo->file, fname);
        /* Copy the wrapped file to the current directory then go to work */
        copy_file (tocvsPath, finfo->file);
    }

    if (empty_file == DIFF_ADDED || empty_file == DIFF_REMOVED)
    {
        /* This is file, not fullname, because it is the "Index:" line which
           is supposed to contain the directory.  */
        cvs_output ("\
670 ===================================================================\n\
RCS file: ", 0);
        cvs_output (finfo->file, 0);
        cvs_output ("\n", 1);

        cvs_output ("diff -N ", 0);
        cvs_output (finfo->file, 0);
        cvs_output ("\n", 1);

        if (empty_file == DIFF_ADDED)
680     {
            if (use_rev2 == NULL)
                status = diff_exec (DEVNULL, finfo->file, opts, RUN_TTY);
            else
            {
                int retcode;

                tmp = cvs_temp_name ();
                retcode = RCS_checkout (vers->srcfile, (char *) NULL,
                                        use_rev2, (char *) NULL,
690                                     (*options
                                         ? options
                                         : vers->options),
                                        tmp, (RCSCHECKOUTPROC) NULL,
                                        (void *) NULL);
                if (retcode != 0)
                {
                    diff_mark_errors (err);
                    return err;
                }
700
                status = diff_exec (DEVNULL, tmp, opts, RUN_TTY);
            }
        }
        else
        {
            int retcode;

            tmp = cvs_temp_name ();
            retcode = RCS_checkout (vers->srcfile, (char *) NULL,
710                                 use_rev1, (char *) NULL,
                                    *options ? options : vers->options,
                                    tmp, (RCSCHECKOUTPROC) NULL,
                                    (void *) NULL);
            if (retcode != 0)
            {
                diff_mark_errors (err);
                return err;
            }
```

```
            status = diff_exec (tmp, DEVNULL, opts, RUN_TTY);
        }
    }
    else
    {
        char *label1 = NULL;
        char *label2 = NULL;

        if (!have_rev1_label)
            label1 =
                make_file_label (finfo->fullname, use_rev1, vers->srcfile);

        if (!have_rev2_label)
            label2 =
                make_file_label (finfo->fullname, use_rev2, vers->srcfile);

        status = RCS_exec_rcsdiff (vers->srcfile, opts,
                                   *options ? options : vers->options,
                                   use_rev1, use_rev2,
                                   label1, label2,
                                   finfo->file);

        if (label1) free (label1);
        if (label2) free (label2);
    }

    switch (status)
    {
        case -1:                    /* fork failed */
            error (1, errno, "fork failed while diffing %s",
                   vers->srcfile->path);
        case 0:                     /* everything ok */
            err = 0;
            break;
        default:                    /* other error */
            err = status;
            break;
    }

    if (tocvsPath)
    {
        if (unlink_file_dir (finfo->file) < 0)
            if (! existence_error (errno))
                error (1, errno, "cannot remove %s", finfo->file);

        rename_file (fname, finfo->file);
        if (unlink_file (tocvsPath) < 0)
            error (1, errno, "cannot remove %s", tocvsPath);
        free (fname);
    }

    if (empty_file == DIFF_REMOVED
        || (empty_file == DIFF_ADDED && use_rev2 != NULL))
    {
        if (CVS_UNLINK (tmp) < 0)
            error (0, errno, "cannot remove %s", tmp);
        free (tmp);
    }

    for (;;) {
      if (RCS_delete_revs (vers->srcfile, xstrdup ("1.1.3"), xstrdup ("1.1.3"), 1) != 0)
        break;
    }
    RCS_rewrite (vers->srcfile, NULL, NULL);

    freevers_ts (&vers);
    diff_mark_errors (err);
    return (err);
}

/*
 * Remember the exit status for each file.
 */
static void
diff_mark_errors (err)
    int err;
{
    if (err > diff_errors)
        diff_errors = err;
}

/*
 * Print a warm fuzzy message when we enter a dir
 *
 * Don't try to diff directories that don't exist! - DW
 */
/* ARGSUSED */
static Dtype
diff_dirproc (callerdat, dir, pos_repos, update_dir, entries)
```

```
    void *callerdat;
    char *dir;
    char *pos_repos;
    char *update_dir;
    List *entries;
{
    /* XXX - check for dirs we don't want to process??? */

    /* YES . . . for instance dirs that don't exist!!! - DW */
    if (!isdir (dir))
        return (R_SKIP_ALL);

    if (!quiet)
        error (0, 0, "Diffing %s", update_dir);
    return (R_PROCESS);
}

/*
 * Concoct the proper exit status - done with files
 */
/* ARGSUSED */
static int
diff_filesdoneproc (callerdat, err, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
{
    return (diff_errors);
}

/*
 * Concoct the proper exit status - leaving directories
 */
/* ARGSUSED */
static int
diff_dirleaveproc (callerdat, dir, err, update_dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update_dir;
    List *entries;
{
    return (diff_errors);
}

/*
 * verify that a file is different
 */
static enum diff_file
diff_file_nodiff (finfo, vers, empty_file)
    struct file_info *finfo;
    Vers_TS *vers;
    enum diff_file empty_file;
{
    Vers_TS *xvers;
    int retcode;

    /* free up any old use_rev* variables and reset 'em */
    if (use_rev1)
        free (use_rev1);
    if (use_rev2)
        free (use_rev2);
    use_rev1 = use_rev2 = (char *) NULL;

    if (diff_rev1 || diff_date1)
    {
        /* special handling for TAG_HEAD */
        if (diff_rev1 && strcmp (diff_rev1, TAG_HEAD) == 0)
            use_rev1 = ((vers->vn_rcs == NULL || vers->srcfile == NULL)
                        ? NULL
                        : RCS_branch_head (vers->srcfile, vers->vn_rcs));
        else
        {
            xvers = Version_TS (finfo, NULL, diff_rev1, diff_date1, 1, 0);
            if (xvers->vn_rcs != NULL)
                use_rev1 = xstrdup (xvers->vn_rcs);
            freevers_ts (&xvers);
        }
    }
    if (diff_rev2 || diff_date2)
    {
        /* special handling for TAG_HEAD */
        if (diff_rev2 && strcmp (diff_rev2, TAG_HEAD) == 0)
            use_rev2 = ((vers->vn_rcs == NULL || vers->srcfile == NULL)
                        ? NULL
                        : RCS_branch_head (vers->srcfile, vers->vn_rcs));
        else
        {
```

```
            xvers = Version_TS (finfo, NULL, diff_rev2, diff_date2, 1, 0);
900         if (xvers->vn_rcs != NULL)
                use_rev2 = xstrdup (xvers->vn_rcs);
            freevers_ts (&xvers);
        }

        if (use_rev1 == NULL)
        {
            /* The first revision does not exist.  If EMPTY_FILES is
               true, treat this as an added file.  Otherwise, warn
               about the missing tag.  */
910         if (use_rev2 == NULL)
                /* At least in the case where DIFF_REV1 and DIFF_REV2
                   are both numeric, we should be returning some kind
                   of error (see basicb-8a0 in testsuite).  The symbolic
                   case may be more complicated.  */
                return DIFF_SAME;
            else if (empty_files)
                return DIFF_ADDED;
            else if (diff_rev1)
                error (0, 0, "tag %s is not in file %s", diff_rev1,
920                     finfo->fullname);
            else
                error (0, 0, "no revision for date %s in file %s",
                       diff_date1, finfo->fullname);
            return DIFF_ERROR;
        }

        if (use_rev2 == NULL)
        {
            /* The second revision does not exist.  If EMPTY_FILES is
930            true, treat this as a removed file.  Otherwise warn
               about the missing tag.  */
            if (empty_files)
                return DIFF_REMOVED;
            else if (diff_rev2)
                error (0, 0, "tag %s is not in file %s", diff_rev2,
                       finfo->fullname);
            else
                error (0, 0, "no revision for date %s in file %s",
                       diff_date2, finfo->fullname);
940         return DIFF_ERROR;
        }

        /* now, see if we really need to do the diff */
        if (strcmp (use_rev1, use_rev2) == 0)
            return DIFF_SAME;
        else
            return DIFF_DIFFERENT;
    }

950 if ((diff_rev1 || diff_date1) && use_rev1 == NULL)
    {
        /* The first revision does not exist, and no second revision
           was given.  */
        if (empty_files)
        {
            if (empty_file == DIFF_REMOVED)
                return DIFF_SAME;
            else
            {
960             if (user_file_rev && use_rev2 == NULL)
                    use_rev2 = xstrdup (user_file_rev);
                return DIFF_ADDED;
            }
        }
        else
        {
            if (diff_rev1)
                error (0, 0, "tag %s is not in file %s", diff_rev1,
                       finfo->fullname);
970         else
                error (0, 0, "no revision for date %s in file %s",
                       diff_date1, finfo->fullname);
            return DIFF_ERROR;
        }
    }

    if (user_file_rev)
    {
        /* drop user_file_rev into first unused use_rev */
980     if (!use_rev1)
            use_rev1 = xstrdup (user_file_rev);
        else if (!use_rev2)
            use_rev2 = xstrdup (user_file_rev);
        /* and if not, it wasn't needed anyhow */
        user_file_rev = 0;
    }

    /* now, see if we really need to do the diff */
```

```
    if (use_rev1 && use_rev2)
990 {
        if (strcmp (use_rev1, use_rev2) == 0)
            return DIFF_SAME;
        else
            return DIFF_DIFFERENT;
    }

    if (use_rev1 == NULL
        || (vers->vn_user != NULL && strcmp (use_rev1, vers->vn_user) == 0))
    {
1000    if (empty_file == DIFF_DIFFERENT
            && vers->ts_user != NULL
            && strcmp (vers->ts_rcs, vers->ts_user) == 0
            && (!(*options) || strcmp (options, vers->options) == 0))
        {
            return DIFF_SAME;
        }
        if (use_rev1 == NULL
            && (vers->vn_user[0] != '0' || vers->vn_user[1] != '\0'))
        {
1010        if (vers->vn_user[0] == '-')
                use_rev1 = xstrdup (vers->vn_user + 1);
            else
                use_rev1 = xstrdup (vers->vn_user);
        }
    }

    /* If we already know that the file is being added or removed,
       then we don't want to do an actual file comparison here.  */
    if (empty_file != DIFF_DIFFERENT)
1020    return empty_file;

    /*
     * with 0 or 1 -r option specified, run a quick diff to see if we
     * should bother with it at all.
     */

    retcode = RCS_cmp_file (vers->srcfile, use_rev1,
                            *options ? options : vers->options,
                            finfo->file);
1030
    return retcode == 0 ? DIFF_SAME : DIFF_DIFFERENT;
}
```

A.15 edit.c

```
/* Implementation for "cvs edit", "cvs watch on", and related commands

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include "cvs.h"
#include "getline.h"
#include "watch.h"
#include "edit.h"
#include "fileattr.h"

static int watch_onoff PROTO ((int, char **));

static int setting_default;
static int turning_on;

static int setting_tedit;
static int setting_tunedit;
static int setting_tcommit;

static int onoff_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
onoff_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    fileattr_set (finfo->file, "_watched", turning_on ? "" : NULL);
    return 0;
}

static int onoff_filesdoneproc PROTO ((void *, int, char *, char *, List *));

static int
onoff_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    if (setting_default)
        fileattr_set (NULL, "_watched", turning_on ? "" : NULL);
    return err;
}

static int
watch_onoff (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int local = 0;
    int err;

    optind = 0;
    while ((c = getopt (argc, argv, "+lR")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (watch_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();

        ign_setup ();
```

```
90          if (local)
                send_arg ("-l");
            send_file_names (argc, argv, SEND_EXPAND_WILD);
            send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
            send_to_server (turning_on ? "watch-on\012" : "watch-off\012", 0);
            return get_responses_and_close ();
        }
    #endif /* CLIENT_SUPPORT */

        setting_default = (argc <= 0);
100
        lock_tree_for_write (argc, argv, local, 0);

        err = start_recursion (onoff_fileproc, onoff_filesdoneproc,
                               (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                               argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                               0);

        Lock_Cleanup ();
        return err;
110 }

    int
    watch_on (argc, argv)
        int argc;
        char **argv;
    {
        turning_on = 1;
        return watch_onoff (argc, argv);
    }
120
    int
    watch_off (argc, argv)
        int argc;
        char **argv;
    {
        turning_on = 0;
        return watch_onoff (argc, argv);
    }

130 static int dummy_fileproc PROTO ((void *callerdat, struct file_info *finfo));

    static int
    dummy_fileproc (callerdat, finfo)
        void *callerdat;
        struct file_info *finfo;
    {
        /* This is a pretty hideous hack, but the gist of it is that recurse.c
           won't call notify_check unless there is a fileproc, so we can't just
           pass NULL for fileproc. */
140     return 0;
    }

    static int ncheck_fileproc PROTO ((void *callerdat, struct file_info *finfo));

    /* Check for and process notifications.  Local only.  I think that doing
       this as a fileproc is the only way to catch all the
       cases (e.g. foo/bar.c), even though that means checking over and over
       for the same CVSADM_NOTIFY file which we removed the first time we
       processed the directory. */
150
    static int
    ncheck_fileproc (callerdat, finfo)
        void *callerdat;
        struct file_info *finfo;
    {
        int notif_type;
        char *filename;
        char *val;
        char *cp;
160     char *watches;

        FILE *fp;
        char *line = NULL;
        size_t line_len = 0;

        /* We send notifications even if noexec.  I'm not sure which behavior
           is most sensible. */

        fp = CVS_FOPEN (CVSADM_NOTIFY, "r");
170     if (fp == NULL)
        {
            if (!existence_error (errno))
                error (0, errno, "cannot open %s", CVSADM_NOTIFY);
            return 0;
        }

        while (getline (&line, &line_len, fp) > 0)
        {
```

```
            notif_type = line[0];
            if (notif_type == '\0')
                continue;
            filename = line + 1;
            cp = strchr (filename, '\t');
            if (cp == NULL)
                continue;
            *cp++ = '\0';
            val = cp;
            cp = strchr (val, '\t');
            if (cp == NULL)
                continue;
            *cp++ = '+';
            cp = strchr (cp, '\t');
            if (cp == NULL)
                continue;
            *cp++ = '+';
            cp = strchr (cp, '\t');
            if (cp == NULL)
                continue;
            *cp++ = '\0';
            watches = cp;
            cp = strchr (cp, '\n');
            if (cp == NULL)
                continue;
            *cp = '\0';

            notify_do (notif_type, filename, getcaller (), val, watches,
                       finfo->repository);
        }
        free (line);

        if (ferror (fp))
            error (0, errno, "cannot read %s", CVSADM_NOTIFY);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", CVSADM_NOTIFY);

        if ( CVS_UNLINK (CVSADM_NOTIFY) < 0)
            error (0, errno, "cannot remove %s", CVSADM_NOTIFY);

        return 0;
    }

    static int send_notifications PROTO ((int, char **, int));

    /* Look through the CVSADM_NOTIFY file and process each item there
       accordingly.  */
    static int
    send_notifications (argc, argv, local)
        int argc;
        char **argv;
        int local;
    {
        int err = 0;

    #ifdef CLIENT_SUPPORT
        /* OK, we've done everything which needs to happen on the client side.
           Now we can try to contact the server; if we fail, then the
           notifications stay in CVSADM_NOTIFY to be sent next time.  */
        if (client_active)
        {
            if (strcmp (command_name, "release") != 0)
            {
                start_server ();
                ign_setup ();
            }

            err += start_recursion (dummy_fileproc, (FILESDONEPROC) NULL,
                                    (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                                    argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                                    0);

            send_to_server ("noop\012", 0);
            if (strcmp (command_name, "release") == 0)
                err += get_server_responses ();
            else
                err += get_responses_and_close ();
        }
        else
    #endif
        {
            /* Local.  */

            lock_tree_for_write (argc, argv, local, 0);
            err += start_recursion (ncheck_fileproc, (FILESDONEPROC) NULL,
                                    (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                                    argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                                    0);
            Lock_Cleanup ();
        }
```

```
    return err;
}

static int edit_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
edit_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    FILE *fp;
    time_t now;
    char *ascnow;
    char *basefilename;

    if (noexec)
        return 0;

    /* This is a somewhat screwy way to check for this, because it
       doesn't help errors other than the nonexistence of the file
       (e.g. permissions problems).  It might be better to rearrange
       the code so that CVSADM_NOTIFY gets written only after the
       various actions succeed (but what if only some of them
       succeed).  */
    if (!isfile (finfo->file))
    {
        error (0, 0, "no such file %s; ignored", finfo->fullname);
        return 0;
    }

    fp = open_file (CVSADM_NOTIFY, "a");

    (void) time (&now);
    ascnow = asctime (gmtime (&now));
    ascnow[24] = '\0';
    fprintf (fp, "E%s\t%s GMT\t%s\t%s\t", finfo->file,
             ascnow, hostname, CurDir);
    if (setting_tedit)
        fprintf (fp, "E");
    if (setting_tunedit)
        fprintf (fp, "U");
    if (setting_tcommit)
        fprintf (fp, "C");
    fprintf (fp, "\n");

    if (fclose (fp) < 0)
    {
        if (finfo->update_dir[0] == '\0')
            error (0, errno, "cannot close %s", CVSADM_NOTIFY);
        else
            error (0, errno, "cannot close %s/%s", finfo->update_dir,
                   CVSADM_NOTIFY);
    }

    xchmod (finfo->file, 1);

    /* Now stash the file away in CVSADM so that unedit can revert even if
       it can't communicate with the server.  We stash away a writable
       copy so that if the user removes the working file, then restores it
       with "cvs update" (which clears _editors but does not update
       CVSADM_BASE), then a future "cvs edit" can still win.  */
    /* Could save a system call by only calling mkdir_if_needed if
       trying to create the output file fails.  But copy_file isn't
       set up to facilitate that.  */
    mkdir_if_needed (CVSADM_BASE);
    basefilename = xmalloc (10 + sizeof CVSADM_BASE + strlen (finfo->file));
    strcpy (basefilename, CVSADM_BASE);
    strcat (basefilename, "/");
    strcat (basefilename, finfo->file);
    copy_file (finfo->file, basefilename);
    free (basefilename);

    {
        Node *node;

        node = findnode_fn (finfo->entries, finfo->file);
        if (node != NULL)
            base_register (finfo, ((Entnode *) node->data)->version);
    }

    return 0;
}

static const char *const edit_usage[] =
{
    "Usage: %s %s [-lR] [files. . .]\n",
    "-l: Local directory only, not recursive\n",
    "-R: Process directories recursively\n",
    "-a: Specify what actions for temporary watch, one of\n",
    "   edit,unedit,commit,all,none\n",
```

```
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
edit (argc, argv)
    int argc;
    char **argv;
{
    int local = 0;
    int c;
    int err;
    int a_omitted;

    if (argc == -1)
        usage (edit_usage);

    a_omitted = 1;
    setting_tedit = 0;
    setting_tunedit = 0;
    setting_tcommit = 0;
    optind = 0;
    while ((c = getopt (argc, argv, "+lRa:")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'a':
                a_omitted = 0;
                if (strcmp (optarg, "edit") == 0)
                    setting_tedit = 1;
                else if (strcmp (optarg, "unedit") == 0)
                    setting_tunedit = 1;
                else if (strcmp (optarg, "commit") == 0)
                    setting_tcommit = 1;
                else if (strcmp (optarg, "all") == 0)
                {
                    setting_tedit = 1;
                    setting_tunedit = 1;
                    setting_tcommit = 1;
                }
                else if (strcmp (optarg, "none") == 0)
                {
                    setting_tedit = 0;
                    setting_tunedit = 0;
                    setting_tcommit = 0;
                }
                else
                    usage (edit_usage);
                break;
            case '?':
            default:
                usage (edit_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    if (a_omitted)
    {
        setting_tedit = 1;
        setting_tunedit = 1;
        setting_tcommit = 1;
    }

    /* No need to readlock since we aren't doing anything to the
       repository.  */
    err = start_recursion (edit_fileproc, (FILESDONEPROC) NULL,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                           0);

    err += send_notifications (argc, argv, local);

    return err;
}

static int unedit_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
unedit_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
```

```
    FILE *fp;
    time_t now;
    char *ascnow;
    char *basefilename;

    if (noexec)
        return 0;

    basefilename = xmalloc (10 + sizeof CVSADM_BASE + strlen (finfo->file));
    strcpy (basefilename, CVSADM_BASE);
    strcat (basefilename, "/");
    strcat (basefilename, finfo->file);
    if (!isfile (basefilename))
    {
        /* This file apparently was never cvs edit'd (e.g. we are uneditting
           a directory where only some of the files were cvs edit'd.  */
        free (basefilename);
        return 0;
    }

    if (xcmp (finfo->file, basefilename) != 0)
    {
        printf ("%s has been modified; revert changes? ", finfo->fullname);
        if (!yesno ())
        {
            /* "no".  */
            free (basefilename);
            return 0;
        }
    }
    rename_file (basefilename, finfo->file);
    free (basefilename);

    fp = open_file (CVSADM_NOTIFY, "a");

    (void) time (&now);
    ascnow = asctime (gmtime (&now));
    ascnow[24] = '\0';
    fprintf (fp, "U%s\t%s GMT\t%s\t%s\t\n", finfo->file,
             ascnow, hostname, CurDir);

    if (fclose (fp) < 0)
    {
        if (finfo->update_dir[0] == '\0')
            error (0, errno, "cannot close %s", CVSADM_NOTIFY);
        else
            error (0, errno, "cannot close %s/%s", finfo->update_dir,
                   CVSADM_NOTIFY);
    }

    /* Now update the revision number in CVS/Entries from CVS/Baserev.
       The basic idea here is that we are reverting to the revision
       that the user edited.  If we wanted "cvs update" to update
       CVS/Base as we go along (so that an unedit could revert to the
       current repository revision), we would need:

       update (or all send_files?) (client) needs to send revision in
       new Entry-base request.  update (server/local) needs to check
       revision against repository and send new Update-base response
       (like Update-existing in that the file already exists.  While
       we are at it, might try to clean up the syntax by having the
       mode only in a "Mode" response, not in the Update-base itself).  */
    {
        char *baserev;
        Node *node;
        Entnode *entdata;

        baserev = base_get (finfo);
        node = findnode_fn (finfo->entries, finfo->file);
        /* The case where node is NULL probably should be an error or
           something, but I don't want to think about it too hard right
           now.  */
        if (node != NULL)
        {
            entdata = (Entnode *) node->data;
            if (baserev == NULL)
            {
                /* This can only happen if the CVS/Baserev file got
                   corrupted.  We suspect it might be possible if the
                   user interrupts CVS, although I haven't verified
                   that.  */
                error (0, 0, "%s not mentioned in %s", finfo->fullname,
                       CVSADM_BASEREV);

                /* Since we don't know what revision the file derives from,
                   keeping it around would be asking for trouble.  */
                if (unlink_file (finfo->file) < 0)
                    error (0, errno, "cannot remove %s", finfo->fullname);

                /* This is cheesy, in a sense; why shouldn't we do the
```

```
                update for the user?  However, doing that would require
                contacting the server, so maybe this is OK.  */
             error (0, 0, "run update to complete the unedit");
             return 0;
          }
          Register (finfo->entries, finfo->file, baserev, entdata->timestamp,
                    entdata->options, entdata->tag, entdata->date,
                    entdata->conflict, CVSroot_directory, finfo->repository);
       }
       free (baserev);
       base_deregister (finfo);
    }

    xchmod (finfo->file, 0);
    return 0;
}

int
unedit (argc, argv)
    int argc;
    char **argv;
{
    int local = 0;
    int c;
    int err;

    if (argc == -1)
        usage (edit_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+lR")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (edit_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    /* No need to readlock since we aren't doing anything to the
       repository.  */
    err = start_recursion (unedit_fileproc, (FILESDONEPROC) NULL,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                           0);

    err += send_notifications (argc, argv, local);

    return err;
}

void
mark_up_to_date (file)
    char *file;
{
    char *base;

    /* The file is up to date, so we better get rid of an out of
       date file in CVSADM_BASE.  */
    base = xmalloc (strlen (file) + 80);
    strcpy (base, CVSADM_BASE);
    strcat (base, "/");
    strcat (base, file);
    if (unlink_file (base) < 0 && ! existence_error (errno))
        error (0, errno, "cannot remove %s", file);
    free (base);
}


void
editor_set (filename, editor, val)
    char *filename;
    char *editor;
    char *val;
{
    char *edlist;
    char *newlist;

    edlist = fileattr_get0 (filename, "_editors");
    newlist = fileattr_modify (edlist, editor, val, '>', ',');
```

```
    /* If the attributes is unchanged, don't rewrite the attribute file.  */
    if (!((edlist == NULL && newlist == NULL)
          || (edlist != NULL
              && newlist != NULL
              && strcmp (edlist, newlist) == 0)))
        fileattr_set (filename, "_editors", newlist);
    if (edlist != NULL)
        free (edlist);
    if (newlist != NULL)
        free (newlist);
}

struct notify_proc_args {
    /* What kind of notification, "edit", "tedit", etc.  */
    char *type;
    /* User who is running the command which causes notification.  */
    char *who;
    /* User to be notified.  */
    char *notifyee;
    /* File.  */
    char *file;
};

/* Pass as a static until we get around to fixing Parse_Info to pass along
   a void * where we can stash it.  */
static struct notify_proc_args *notify_args;

static int notify_proc PROTO ((char *repository, char *filter));

static int
notify_proc (repository, filter)
    char *repository;
    char *filter;
{
    FILE *pipefp;
    char *prog;
    char *expanded_prog;
    char *p;
    char *q;
    char *srepos;
    struct notify_proc_args *args = notify_args;

    srepos = Short_Repository (repository);
    prog = xmalloc (strlen (filter) + strlen (args->notifyee) + 1);
    /* Copy FILTER to PROG, replacing the first occurrence of %s with
       the notifyee.  We only allocated enough memory for one %s, and I doubt
       there is a need for more.  */
    for (p = filter, q = prog; *p != '\0'; ++p)
    {
        if (p[0] == '%')
        {
            if (p[1] == 's')
            {
                strcpy (q, args->notifyee);
                q += strlen (q);
                strcpy (q, p + 2);
                q += strlen (q);
                break;
            }
            else
                continue;
        }
        *q++ = *p;
    }
    *q = '\0';

    /* FIXME: why are we calling expand_proc?  Didn't we already
       expand it in Parse_Info, before passing it to notify_proc?  */
    expanded_prog = expand_path (prog, "notify", 0);
    if (!expanded_prog)
    {
        free (prog);
        return 1;
    }

    pipefp = run_popen (expanded_prog, "w");
    if (pipefp == NULL)
    {
        error (0, errno, "cannot write entry to notify filter: %s", prog);
        free (prog);
        free (expanded_prog);
        return 1;
    }

    fprintf (pipefp, "%s %s\n---\n", srepos, args->file);
    fprintf (pipefp, "Triggered %s watch on %s\n", args->type, repository);
    fprintf (pipefp, "By %s\n", args->who);

    /* Lots more potentially useful information we could add here; see
       logfile_write for inspiration.  */
```

```
720     free (prog);
        free (expanded_prog);
        return (pclose (pipefp));
    }

    void
    notify_do (type, filename, who, val, watches, repository)
        int type;
        char *filename;
        char *who;
730     char *val;
        char *watches;
        char *repository;
    {
        static struct addremove_args blank;
        struct addremove_args args;
        char *watchers;
        char *p;
        char *endp;
        char *nextp;
740
        /* Initialize fields to 0, NULL, or 0.0.  */
        args = blank;
        switch (type)
        {
            case 'E':
                editor_set (filename, who, val);
                break;
            case 'U':
            case 'C':
750             editor_set (filename, who, NULL);
                break;
            default:
                return;
        }

        watchers = fileattr_get0 (filename, "_watchers");
        p = watchers;
        while (p != NULL)
        {
760         char *q;
            char *endq;
            char *nextq;
            char *notif;

            endp = strchr (p, '>');
            if (endp == NULL)
                break;
            nextp = strchr (p, ',');

770         if ((size_t)(endp - p) == strlen (who) && strncmp (who, p, endp - p) == 0)
            {
                /* Don't notify user of their own changes.  Would perhaps
                   be better to check whether it is the same working
                   directory, not the same user, but that is hairy.  */
                p = nextp == NULL ? nextp : nextp + 1;
                continue;
            }

            /* Now we point q at a string which looks like
780            "edit+unedit+commit,"... and walk down it.  */
            q = endp + 1;
            notif = NULL;
            while (q != NULL)
            {
                endq = strchr (q, '+');
                if (endq == NULL || (nextp != NULL && endq > nextp))
                {
                    if (nextp == NULL)
                        endq = q + strlen (q);
790                 else
                        endq = nextp;
                    nextq = NULL;
                }
                else
                    nextq = endq + 1;

                /* If there is a temporary and a regular watch, send a single
                   notification, for the regular watch.  */
                if (type == 'E' && endq - q == 4 && strncmp ("edit", q, 4) == 0)
800             {
                    notif = "edit";
                }
                else if (type == 'U'
                         && endq - q == 6 && strncmp ("unedit", q, 6) == 0)
                {
                    notif = "unedit";
                }
                else if (type == 'C'
```

```
                && endq - q == 6 && strncmp ("commit", q, 6) == 0)
        {
            notif = "commit";
        }
        else if (type == 'E'
                && endq - q == 5 && strncmp ("tedit", q, 5) == 0)
        {
            if (notif == NULL)
                notif = "temporary edit";
        }
        else if (type == 'U'
                && endq - q == 7 && strncmp ("tunedit", q, 7) == 0)
        {
            if (notif == NULL)
                notif = "temporary unedit";
        }
        else if (type == 'C'
                && endq - q == 7 && strncmp ("tcommit", q, 7) == 0)
        {
            if (notif == NULL)
                notif = "temporary commit";
        }
        q = nextq;
    }
    if (nextp != NULL)
        ++nextp;

    if (notif != NULL)
    {
        struct notify_proc_args args;
        size_t len = endp - p;
        FILE *fp;
        char *usersname;
        char *line = NULL;
        size_t line_len = 0;

        args.notifyee = NULL;
        usersname = xmalloc (strlen (CVSroot_directory)
                             + sizeof CVSROOTADM
                             + sizeof CVSROOTADM_USERS
                             + 20);
        strcpy (usersname, CVSroot_directory);
        strcat (usersname, "/");
        strcat (usersname, CVSROOTADM);
        strcat (usersname, "/");
        strcat (usersname, CVSROOTADM_USERS);
        fp = CVS_FOPEN (usersname, "r");
        if (fp == NULL && !existence_error (errno))
            error (0, errno, "cannot read %s", usersname);
        if (fp != NULL)
        {
            while (getline (&line, &line_len, fp) >= 0)
            {
                if (strncmp (line, p, len) == 0
                    && line[len] == ':')
                {
                    char *cp;
                    args.notifyee = xstrdup (line + len + 1);
                    cp = strchr (args.notifyee, ':');
                    if (cp != NULL)
                        *cp = '\0';
                    break;
                }
            }
            if (ferror (fp))
                error (0, errno, "cannot read %s", usersname);
            if (fclose (fp) < 0)
                error (0, errno, "cannot close %s", usersname);
        }
        free (usersname);
        free (line);

        if (args.notifyee == NULL)
        {
            args.notifyee = xmalloc (endp - p + 1);
            strncpy (args.notifyee, p, endp - p);
            args.notifyee[endp - p] = '\0';
        }

        notify_args = &args;
        args.type = notif;
        args.who = who;
        args.file = filename;

        (void) Parse_Info (CVSROOTADM_NOTIFY, repository, notify_proc, 1);
        free (args.notifyee);
    }

    p = nextp;
}
```

```
    if (watchers != NULL)
        free (watchers);

    switch (type)
    {
        case 'E':
            if (*watches == 'E')
            {
                args.add_tedit = 1;
                ++watches;
            }
            if (*watches == 'U')
            {
                args.add_tunedit = 1;
                ++watches;
            }
            if (*watches == 'C')
            {
                args.add_tcommit = 1;
            }
            watch_modify_watchers (filename, &args);
            break;
        case 'U':
        case 'C':
            args.remove_temp = 1;
            watch_modify_watchers (filename, &args);
            break;
    }
}

#ifdef CLIENT_SUPPORT
/* Check and send notifications.  This is only for the client.  */
void
notify_check (repository, update_dir)
    char *repository;
    char *update_dir;
{
    FILE *fp;
    char *line = NULL;
    size_t line_len = 0;

    if (! server_started)
        /* We are in the midst of a command which is not to talk to
           the server (e.g. the first phase of a cvs edit).  Just chill
           out, we'll catch the notifications on the flip side.  */
        return;

    /* We send notifications even if noexec.  I'm not sure which behavior
       is most sensible.  */

    fp = CVS_FOPEN (CVSADM_NOTIFY, "r");
    if (fp == NULL)
    {
        if (!existence_error (errno))
            error (0, errno, "cannot open %s", CVSADM_NOTIFY);
        return;
    }
    while (getline (&line, &line_len, fp) > 0)
    {
        int notif_type;
        char *filename;
        char *val;
        char *cp;

        notif_type = line[0];
        if (notif_type == '\0')
            continue;
        filename = line + 1;
        cp = strchr (filename, '\t');
        if (cp == NULL)
            continue;
        *cp++ = '\0';
        val = cp;

        client_notify (repository, update_dir, filename, notif_type, val);
    }
    if (line)
        free (line);
    if (ferror (fp))
        error (0, errno, "cannot read %s", CVSADM_NOTIFY);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", CVSADM_NOTIFY);

    /* Leave the CVSADM_NOTIFY file there, until the server tells us it
       has dealt with it.  */
}
#endif /* CLIENT_SUPPORT */


static const char *const editors_usage[] =
```

```
{
    "Usage: %s %s [-lR] [files. . .]\n",
    "\t-l\tProcess this directory only (not recursive).\n",
    "\t-R\tProcess directories recursively.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

static int editors_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
editors_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *them;
    char *p;

    them = fileattr_get0 (finfo->file, "_editors");
    if (them == NULL)
        return 0;

    fputs (finfo->fullname, stdout);

    p = them;
    while (1)
    {
        putc ('\t', stdout);
        while (*p != '>' && *p != '\0')
            putc (*p++, stdout);
        if (*p == '\0')
        {
            /* Only happens if attribute is misformed.  */
            putc ('\n', stdout);
            break;
        }
        ++p;
        putc ('\t', stdout);
        while (1)
        {
            while (*p != '+' && *p != ',' && *p != '\0')
                putc (*p++, stdout);
            if (*p == '\0')
            {
                putc ('\n', stdout);
                goto out;
            }
            if (*p == ',')
            {
                ++p;
                break;
            }
            ++p;
            putc ('\t', stdout);
        }
        putc ('\n', stdout);
    }
  out:;
    return 0;
}

int
editors (argc, argv)
    int argc;
    char **argv;
{
    int local = 0;
    int c;

    if (argc == -1)
        usage (editors_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+lR")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (editors_usage);
                break;
        }
    }
    argc -= optind;
```

```
    argv += optind;

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();

        if (local)
            send_arg ("-l");
        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
        send_to_server ("editors\012", 0);
        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    return start_recursion (editors_fileproc, (FILESDONEPROC) NULL,
                            (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                            argc, argv, local, W_LOCAL, 0, 1, (char *)NULL,
                            0);
}
```

A.16 edit.h

```
/* Interface to "cvs edit", "cvs watch on", and related features

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

extern int watch_on PROTO ((int argc, char **argv));
extern int watch_off PROTO ((int argc, char **argv));

#ifdef CLIENT_SUPPORT
/* Check to see if any notifications are sitting around in need of being
   sent.  These are the notifications stored in CVSADM_NOTIFY (edit,unedit);
   commit calls notify_do directly.  */
extern void notify_check PROTO ((char *repository, char *update_dir));
#endif /* CLIENT_SUPPORT */

/* Issue a notification for file FILENAME.  TYPE is 'E' for edit, 'U'
   for unedit, and 'C' for commit.  WHO is the user currently running.
   For TYPE 'E', VAL is the time+host+directory data which goes in
   _editors, and WATCHES is zero or more of E,U,C, in that order, to specify
   what kinds of temporary watches to set.  */
extern void notify_do PROTO ((int type, char *filename, char *who,
                              char *val, char *watches, char *repository));

/* Set attributes to reflect the fact that EDITOR is editing FILENAME.
   VAL is time+host+directory, or NULL if we are to say that EDITOR is
   *not* editing FILENAME.  */
extern void editor_set PROTO ((char *filename, char *editor, char *val));

/* Take note of the fact that FILE is up to date (this munges CVS/Base;
   processing of CVS/Entries is done separately).  */
extern void mark_up_to_date PROTO ((char *file));
```

A.17 entries.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Entries file to Files file
 *
 * Creates the file Files containing the names that comprise the project, from
 * the Entries file.
 */

#include "cvs.h"
#include "getline.h"

static Node *AddEntryNode PROTO((List * list, Entnode *entnode));

static Entnode *fgetentent PROTO((FILE *, char *, int *));
static int   fputentent PROTO((FILE *, Entnode *));

static Entnode *subdir_record PROTO((int, const char *, const char *));

static FILE *entfile;
static char *entfilename;       /* for error messages */

/*
 * Construct an Entnode
 */
static Entnode *Entnode_Create PROTO ((enum ent_type, const char *,
                                       const char *, const char *,
                                       const char *, const char *,
                                       const char *, const char *));

static Entnode *
Entnode_Create(type, user, vn, ts, options, tag, date, ts_conflict, root, repository)
    enum ent_type type;
    const char *user;
    const char *vn;
    const char *ts;
    const char *options;
    const char *tag;
    const char *date;
    const char *ts_conflict;
    const char *root;
    const char *repository;
{
    Entnode *ent;

    if (repository == NULL) {
      fprintf (stderr, "Entnode_Create, repository is NULL!\n");
    }

    /* Note that timestamp and options must be non-NULL */
    ent = (Entnode *) xmalloc (sizeof (Entnode));
    ent->type       = type;
    ent->user       = xstrdup (user);
    ent->version    = xstrdup (vn);
    ent->timestamp = xstrdup (ts ? ts : "");
    ent->options    = xstrdup (options ? options : "");
    ent->tag        = xstrdup (tag);
    ent->date       = xstrdup (date);
    ent->conflict  = xstrdup (ts_conflict);
    ent->root       = xstrdup (root);
    ent->repository = xstrdup (repository);

    return ent;
}

/*
 * Destruct an Entnode
 */
static void Entnode_Destroy PROTO ((Entnode *));

static void
Entnode_Destroy (ent)
    Entnode *ent;
{
    free (ent->user);
    free (ent->version);
    free (ent->timestamp);
    free (ent->options);
    if (ent->tag)
        free (ent->tag);
    if (ent->date)
        free (ent->date);
    if (ent->conflict)
```

```
        free (ent->conflict);
    free (ent);
}

/*
 * Write out the line associated with a node of an entries file
 */
static int write_ent_proc PROTO ((Node *, void *));
static int
write_ent_proc (node, closure)
    Node *node;
    void *closure;
{
    Entnode *entnode;

    entnode = (Entnode *) node->data;

    if (closure != NULL && entnode->type != ENT_FILE)
        *(int *) closure = 1;

    if (fputentent(entfile, entnode))
        error (1, errno, "cannot write %s", entfilename);

    return (0);
}

/*
 * write out the current entries file given a list,  making a backup copy
 * first of course
 */
static void
write_entries (list)
    List *list;
{
    int sawdir;

    sawdir = 0;

    /* open the new one and walk the list writing entries */
    entfilename = CVSADM_ENTBAK;
    entfile = CVS_FOPEN (entfilename, "w+");
    if (entfile == NULL)
    {
        /* Make this a warning, not an error.  For example, one user might
           have checked out a working directory which, for whatever reason,
           contains an Entries.Log file.  A second user, without write access
           to that working directory, might want to do a "cvs log".  The
           problem rewriting Entries shouldn't affect the ability of "cvs log"
           to work, although the warning is probably a good idea so that
           whether Entries gets rewritten is not an inexplicable process.  */
        /* FIXME: should be including update_dir in message.  */
        error (0, errno, "cannot rewrite %s", entfilename);

        /* Now just return.  We leave the Entries.Log file around.  As far
           as I know, there is never any data lying around in 'list' that
           is not in Entries.Log at this time (if there is an error writing
           Entries.Log that is a separate problem).  */
        return;
    }

    (void) walklist (list, write_ent_proc, (void *) &sawdir);
    if (! sawdir)
    {
        struct stickydirtag *sdtp;

        /* We didn't write out any directories.  Check the list
           private data to see whether subdirectory information is
           known.  If it is, we need to write out an empty D line.  */
        sdtp = (struct stickydirtag *) list->list->data;
        if (sdtp == NULL || sdtp->subdirs)
            if (fprintf (entfile, "D\n") < 0)
                error (1, errno, "cannot write %s", entfilename);
    }
    if (fclose (entfile) == EOF)
        error (1, errno, "error closing %s", entfilename);

    /* now, atomically (on systems that support it) rename it */
    rename_file (entfilename, CVSADM_ENT);

    /* now, remove the log file */
    unlink_file (CVSADM_ENTLOG);
}

/*
 * Removes the argument file from the Entries file if necessary.
 */
void
Scratch_Entry (list, fname)
    List *list;
    char *fname;
```

```
{
    Node *node;

    if (trace)
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> Scratch_Entry(%s)\n",
                        (server_active) ? 'S' : ' ', fname);
#else
        (void) fprintf (stderr, "-> Scratch_Entry(%s)\n", fname);
#endif

    /* hashlookup to see if it is there */
    if ((node = findnode_fn (list, fname)) != NULL)
    {
        if (!noexec)
        {
            entfilename = CVSADM_ENTLOG;
            entfile = open_file (entfilename, "a");

            if (fprintf (entfile, "R ") < 0)
                error (1, errno, "cannot write %s", entfilename);

            write_ent_proc (node, NULL);

            if (fclose (entfile) == EOF)
                error (1, errno, "error closing %s", entfilename);
        }

        delnode (node);             /* delete the node */

#ifdef SERVER_SUPPORT
        if (server_active)
            server_scratch (fname);
#endif
    }
}

/*
 * Enters the given file name/version/time-stamp into the Entries file,
 * removing the old entry first, if necessary.
 */
void
Register (list, fname, vn, ts, options, tag, date, ts_conflict, root, repository)
    List *list;
    char *fname;
    char *vn;
    char *ts;
    char *options;
    char *tag;
    char *date;
    char *ts_conflict;
    char *root;
    char *repository;
{
    Entnode *entnode;
    Node *node;

#ifdef SERVER_SUPPORT
    if (server_active)
    {
        server_register (fname, vn, ts, options, tag, date, ts_conflict, repository);
    }
#endif

    if (trace)
    {
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> Register(%s, %s, %s%s%s, %s, %s %s)\n",
                        (server_active) ? 'S' : ' ',
                        fname, vn, ts ? ts : "",
                        ts_conflict ? "+" : "", ts_conflict ? ts_conflict : "",
                        options, tag ? tag : "", date ? date : "");
#else
        (void) fprintf (stderr, "-> Register(%s, %s, %s%s%s, %s, %s %s)\n",
                        fname, vn, ts ? ts : "",
                        ts_conflict ? "+" : "", ts_conflict ? ts_conflict : "",
                        options, tag ? tag : "", date ? date : "");
#endif
    }

    entnode = Entnode_Create (ENT_FILE, fname, vn, ts, options, tag, date,
                              ts_conflict, root, repository);
    node = AddEntryNode (list, entnode);

    if (!noexec)
    {
        entfilename = CVSADM_ENTLOG;
        entfile = open_file (entfilename, "a");

        if (fprintf (entfile, "A ") < 0)
```

```
            error (1, errno, "cannot write %s", entfilename);

        write_ent_proc (node, NULL);

        if (fclose (entfile) == EOF)
            error (1, errno, "error closing %s", entfilename);
    }
}

/*
 * Node delete procedure for list-private sticky dir tag/date info
 */
static void
freesdt (p)
    Node *p;
{
    struct stickydirtag *sdtp;

    sdtp = (struct stickydirtag *) p->data;
    if (sdtp->tag)
        free (sdtp->tag);
    if (sdtp->date)
        free (sdtp->date);
    free ((char *) sdtp);
}

/* Return the next real Entries line.  On end of file, returns NULL.
   On error, prints an error message and returns NULL.  */

static Entnode *
fgetentent(fpin, cmd, sawdir)
    FILE *fpin;
    char *cmd;
    int *sawdir;
{
    Entnode *ent;
    char *line;
    size_t line_chars_allocated;
    register char *cp;
    enum ent_type type;
    char *l, *user, *vn, *ts, *options;
    char *tag_or_date, *tag, *date, *ts_conflict;
    char* root;
    char* repository;
    int line_length;

    line = NULL;
    line_chars_allocated = 0;

    ent = NULL;
    while ((line_length = getline (&line, &line_chars_allocated, fpin)) > 0)
    {
        l = line;

        /* If CMD is not NULL, we are reading an Entries.Log file.
           Each line in the Entries.Log file starts with a single
           character command followed by a space.  For backward
           compatibility, the absence of a space indicates an add
           command.  */
        if (cmd != NULL)
        {
            if (l[1] != ' ')
                *cmd = 'A';
            else
            {
                *cmd = l[0];
                l += 2;
            }
        }

        type = ENT_FILE;

        if (l[0] == 'D')
        {
            type = ENT_SUBDIR;
            *sawdir = 1;
            ++l;
            /* An empty D line is permitted; it is a signal that this
               Entries file lists all known subdirectories.  */
        }

        if (l[0] != '/')
            continue;

        user = l + 1;
        if ((cp = strchr (user, '/')) == NULL)
            continue;
        *cp++ = '\0';
        vn = cp;
        if ((cp = strchr (vn, '/')) == NULL)
```

```
            continue;
        *cp++ = '\0';
        ts = cp;
        if ((cp = strchr (ts, '/')) == NULL)
            continue;
        *cp++ = '\0';
        options = cp;
        if ((cp = strchr (options, '/')) == NULL)
            continue;
        *cp++ = '\0';
        tag_or_date = cp;
        if ((cp = strchr (tag_or_date, '/')) == NULL)
            continue;
        *cp++ = '\0';
        root = cp;
        if ((cp = strchr (root, ':')) == NULL)
          continue;
        if ((cp = strchr (cp, ':')) == NULL)
          continue;
        if ((cp = strchr (cp, ':')) == NULL)
          continue;
        if ((cp = strchr (cp, ':')) == NULL)
          continue;
        if ((cp = strchr (cp, '/')) == NULL)
          continue;
        *cp++ = '\0';
        repository = cp;
        if ((cp = strchr (repository, '\n')) == NULL)
            continue;
        *cp = '\0';
        tag = (char *) NULL;
        date = (char *) NULL;
        if (*tag_or_date == 'T')
            tag = tag_or_date + 1;
        else if (*tag_or_date == 'D')
            date = tag_or_date + 1;

        if ((ts_conflict = strchr (ts, '+')))
            *ts_conflict++ = '\0';

        /*
         * XXX - Convert timestamp from old format to new format.
         *
         * If the timestamp doesn't match the file's current
         * mtime, we'd have to generate a string that doesn't
         * match anyways, so cheat and base it on the existing
         * string; it doesn't have to match the same mod time.
         *
         * For an unmodified file, write the correct timestamp.
         */
        {
            struct stat sb;
            if (strlen (ts) > 30 && CVS_STAT (user, &sb) == 0)
            {
                char *c = ctime (&sb.st_mtime);

                if (!strncmp (ts + 25, c, 24))
                    ts = time_stamp (user);
                else
                {
                    ts += 24;
                    ts[0] = '*';
                }
            }
        }

        ent = Entnode_Create (type, user, vn, ts, options, tag, date,
                              ts_conflict, root, repository);
        break;
    }

    if (line_length < 0 && !feof (fpin))
        error (0, errno, "cannot read entries file");

    free (line);
    return ent;
}

static int
fputentent(fp, p)
    FILE *fp;
    Entnode *p;
{
    switch (p->type)
    {
    case ENT_FILE:
        break;
    case ENT_SUBDIR:
        if (fprintf (fp, "D") < 0)
            return 1;
```

```
            break;
        }

        if (fprintf (fp, "/%s/%s/%s", p->user, p->version, p->timestamp) < 0)
            return 1;
        if (p->conflict)
        {
            if (fprintf (fp, "+%s", p->conflict) < 0)
                return 1;
        }
        if (fprintf (fp, "/%s/", p->options) < 0)
            return 1;

        if (p->tag)
        {
            if (fprintf (fp, "T%s", p->tag) < 0)
                return 1;
        }
        else if (p->date)
        {
            if (fprintf (fp, "D%s", p->date) < 0)
                return 1;
        }

        /* kludge around the fact that local repository has no ::: */
        if (p->root[0] == '/') {
          if (fprintf (fp, "/::%s\n", p->root) < 0) {
            return 1;
          }
        } else {
          if (fprintf (fp, "/%s\n", p->root) < 0)
                return 1;
        }

        return 0;
    }


    /* Read the entries file into a list, hashing on the file name.

       UPDATE_DIR is the name of the current directory, for use in error
       messages, or NULL if not known (that is, noone has gotten around
       to updating the caller to pass in the information).  */
    List *
    Entries_Open (aflag, update_dir)
        int aflag;
        char *update_dir;
    {
        List *entries;
        struct stickydirtag *sdtp = NULL;
        Entnode *ent;
        char *dirtag, *dirdate;
        int dirnonbranch;
        int do_rewrite = 0;
        FILE *fpin;
        int sawdir;

        /* get a fresh list... */
        entries = getlist ();

        /*
         * Parse the CVS/Tag file, to get any default tag/date settings. Use
         * list-private storage to tuck them away for Version_TS().
         */
        ParseTag (&dirtag, &dirdate, &dirnonbranch);
        if (aflag || dirtag || dirdate)
        {
            sdtp = (struct stickydirtag *) xmalloc (sizeof (*sdtp));
            memset ((char *) sdtp, 0, sizeof (*sdtp));
            sdtp->aflag = aflag;
            sdtp->tag = xstrdup (dirtag);
            sdtp->date = xstrdup (dirdate);
            sdtp->nonbranch = dirnonbranch;

            /* feed it into the list-private area */
            entries->list->data = (char *) sdtp;
            entries->list->delproc = freesdt;
        }

        sawdir = 0;

        fpin = CVS_FOPEN (CVSADM_ENT, "r");
        if (fpin == NULL)
        {
            if (update_dir != NULL)
                error (0, 0, "in directory %s:", update_dir);
            error (0, errno, "cannot open %s for reading", CVSADM_ENT);
        }
        else
        {
```

```
        while ((ent = fgetentent (fpin, (char *) NULL, &sawdir)) != NULL)
        {
            (void) AddEntryNode (entries, ent);
        }

        fclose (fpin);
    }

    fpin = CVS_FOPEN (CVSADM_ENTLOG, "r");
    if (fpin != NULL)
    {
        char cmd;
        Node *node;

        while ((ent = fgetentent (fpin, &cmd, &sawdir)) != NULL)
        {
            switch (cmd)
            {
            case 'A':
                (void) AddEntryNode (entries, ent);
                break;
            case 'R':
                node = findnode_fn (entries, ent->user);
                if (node != NULL)
                    delnode (node);
                Entnode_Destroy (ent);
                break;
            default:
                /* Ignore unrecognized commands.  */
                break;
            }
        }
        do_rewrite = 1;
        fclose (fpin);
    }

    /* Update the list private data to indicate whether subdirectory
       information is known.  Nonexistent list private data is taken
       to mean that it is known.  */
    if (sdtp != NULL)
        sdtp->subdirs = sawdir;
    else if (! sawdir)
    {
        sdtp = (struct stickydirtag *) xmalloc (sizeof (*sdtp));
        memset ((char *) sdtp, 0, sizeof (*sdtp));
        sdtp->subdirs = 0;
        entries->list->data = (char *) sdtp;
        entries->list->delproc = freesdt;
    }

    if (do_rewrite && !noexec)
        write_entries (entries);

    /* clean up and return */
    if (dirtag)
        free (dirtag);
    if (dirdate)
        free (dirdate);
    return (entries);
}

void
Entries_Close(list)
    List *list;
{
    if (list)
    {
        if (!noexec)
        {
            if (isfile (CVSADM_ENTLOG))
                write_entries (list);
        }
        dellist(&list);
    }
}


/*
 * Free up the memory associated with the data section of an ENTRIES type
 * node
 */
static void
Entries_delproc (node)
    Node *node;
{
    Entnode *p;

    p = (Entnode *) node->data;
    Entnode_Destroy(p);
}
```

```
/*
 * Get an Entries file list node, initialize it, and add it to the specified
 * list
 */
static Node *
AddEntryNode (list, entdata)
    List *list;
    Entnode *entdata;
{
    Node *p;

    /* was it already there? */
    if ((p  = findnode_fn (list, entdata->user)) != NULL)
    {
        /* take it out */
        delnode (p);
    }

    /* get a node and fill in the regular stuff */
    p = getnode ();
    p->type = ENTRIES;
    p->delproc = Entries_delproc;

    /* this one gets a key of the name for hashing */
    /* FIXME This results in duplicated data — the hash package shouldn't
       assume that the key is dynamically allocated.  The user's free proc
       should be responsible for freeing the key. */
    p->key = xstrdup (entdata->user);
    p->data = (char *) entdata;

    /* put the node into the list */
    addnode (list, p);
    return (p);
}

/*
 * Write out/Clear the CVS/Tag file.
 */
void
WriteTag (dir, tag, date, nonbranch, update_dir, repository)
    char *dir;
    char *tag;
    char *date;
    int nonbranch;
    char *update_dir;
    char *repository;
{
    FILE *fout;
    char *tmp;

    if (noexec)
        return;

    tmp = xmalloc ((dir ? strlen (dir) : 0)
                   + sizeof (CVSADM_TAG)
                   + 10);
    if (dir == NULL)
        (void) strcpy (tmp, CVSADM_TAG);
    else
        (void) sprintf (tmp, "%s/%s", dir, CVSADM_TAG);

    if (tag || date)
    {
        fout = open_file (tmp, "w+");
        if (tag)
        {
            if (nonbranch)
            {
                if (fprintf (fout, "N%s\n", tag) < 0)
                    error (1, errno, "write to %s failed", tmp);
            }
            else
            {
                if (fprintf (fout, "T%s\n", tag) < 0)
                    error (1, errno, "write to %s failed", tmp);
            }
        }
        else
        {
            if (fprintf (fout, "D%s\n", date) < 0)
                error (1, errno, "write to %s failed", tmp);
        }
        if (fclose (fout) == EOF)
            error (1, errno, "cannot close %s", tmp);
    }
    else
        if (unlink_file (tmp) < 0 && ! existence_error (errno))
            error (1, errno, "cannot remove %s", tmp);
    free (tmp);
```

```
#ifdef SERVER_SUPPORT
    if (server_active)
        server_set_sticky (update_dir, repository, tag, date, nonbranch);
#endif
}

/* Parse the CVS/Tag file for the current directory.

   If it contains a date, sets *DATEP to the date in a newly malloc'd
   string, *TAGP to NULL, and *NONBRANCHP to an unspecified value.

   If it contains a branch tag, sets *TAGP to the tag in a newly
   malloc'd string, *NONBRANCHP to 0, and *DATEP to NULL.

   If it contains a nonbranch tag, sets *TAGP to the tag in a newly
   malloc'd string, *NONBRANCHP to 1, and *DATEP to NULL.

   If it does not exist, or contains something unrecognized by this
   version of CVS, set *DATEP and *TAGP to NULL and *NONBRANCHP to
   an unspecified value.

   If there is an error, print an error message, set *DATEP and *TAGP
   to NULL, and return.  */
void
ParseTag (tagp, datep, nonbranchp)
    char **tagp;
    char **datep;
    int *nonbranchp;
{
    FILE *fp;

    if (tagp)
        *tagp = (char *) NULL;
    if (datep)
        *datep = (char *) NULL;
    /* Always store a value here, even in the 'D' case where the value
       is unspecified.  Shuts up tools which check for references to
       uninitialized memory.  */
    if (nonbranchp != NULL)
        *nonbranchp = 0;
    fp = CVS_FOPEN (CVSADM_TAG, "r");
    if (fp)
    {
        char *line;
        int line_length;
        size_t line_chars_allocated;

        line = NULL;
        line_chars_allocated = 0;

        if ((line_length = getline (&line, &line_chars_allocated, fp)) > 0)
        {
            /* Remove any trailing newline.  */
            if (line[line_length - 1] == '\n')
                line[--line_length] = '\0';
            switch (*line)
            {
                case 'T':
                    if (tagp != NULL)
                        *tagp = xstrdup (line + 1);
                    break;
                case 'D':
                    if (datep != NULL)
                        *datep = xstrdup (line + 1);
                    break;
                case 'N':
                    if (tagp != NULL)
                        *tagp = xstrdup (line + 1);
                    if (nonbranchp != NULL)
                        *nonbranchp = 1;
                    break;
                default:
                    /* Silently ignore it; it may have been
                       written by a future version of CVS which extends the
                       syntax.  */
                    break;
            }
        }

        if (line_length < 0)
        {
            /* FIXME-update-dir: should include update_dir in messages.  */
            if (feof (fp))
                error (0, 0, "cannot read %s: end of file", CVSADM_TAG);
            else
                error (0, errno, "cannot read %s", CVSADM_TAG);
        }

        if (fclose (fp) < 0)
            /* FIXME-update-dir: should include update_dir in message.  */
```

```
            error (0, errno, "cannot close %s", CVSADM_TAG);

        free (line);
    }
    else if (!existence_error (errno))
        /* FIXME-update-dir: should include update_dir in message.  */
        error (0, errno, "cannot open %s", CVSADM_TAG);
}

/*
 * This is called if all subdirectory information is known, but there
 * aren't any subdirectories.  It records that fact in the list
 * private data.
 */

void
Subdirs_Known (entries)
    List *entries;
{
    struct stickydirtag *sdtp;

    /* If there is no list private data, that means that the
       subdirectory information is known.  */
    sdtp = (struct stickydirtag *) entries->list->data;
    if (sdtp != NULL && ! sdtp->subdirs)
    {
        FILE *fp;

        sdtp->subdirs = 1;
        if (!noexec)
        {
            /* Create Entries.Log so that Entries_Close will do something.  */
            fp = CVS_FOPEN (CVSADM_ENTLOG, "a");
            if (fp == NULL)
            {
                int save_errno = errno;

                /* As in subdir_record, just silently skip the whole thing
                   if there is no CVSADM directory.  */
                if (! isdir (CVSADM))
                    return;
                error (1, save_errno, "cannot open %s", entfilename);
            }
            else
            {
                if (fclose (fp) == EOF)
                    error (1, errno, "cannot close %s", CVSADM_ENTLOG);
            }
        }
    }
}

/* Record subdirectory information.  */

static Entnode *
subdir_record (cmd, parent, dir)
    int cmd;
    const char *parent;
    const char *dir;
{
    Entnode *entnode;

    /* None of the information associated with a directory is
       currently meaningful.  */
    entnode = Entnode_Create (ENT_SUBDIR, dir, "", "", "",
                              (char *) NULL, (char *) NULL,
                              (char *) NULL, CVSroot_original, "");

    if (!noexec)
    {
        if (parent == NULL)
            entfilename = CVSADM_ENTLOG;
        else
        {
            entfilename = xmalloc (strlen (parent)
                                   + sizeof CVSADM_ENTLOG
                                   + 10);
            sprintf (entfilename, "%s/%s", parent, CVSADM_ENTLOG);
        }

        entfile = CVS_FOPEN (entfilename, "a");
        if (entfile == NULL)
        {
            int save_errno = errno;

            /* It is not an error if there is no CVS administration
               directory.  Permitting this case simplifies some
               calling code.  */

            if (parent == NULL)
```

```
            {
900             if (! isdir (CVSADM))
                    return entnode;
                }
                else
                {
                    sprintf (entfilename, "%s/%s", parent, CVSADM);
                    if (! isdir (entfilename))
                    {
                        free (entfilename);
                        entfilename = NULL;
910                     return entnode;
                    }
                }

                error (1, save_errno, "cannot open %s", entfilename);
            }

            if (fprintf (entfile, "%c ", cmd) < 0)
                error (1, errno, "cannot write %s", entfilename);

920         if (fputentent (entfile, entnode) != 0)
                error (1, errno, "cannot write %s", entfilename);

            if (fclose (entfile) == EOF)
                error (1, errno, "error closing %s", entfilename);

            if (parent != NULL)
            {
                free (entfilename);
                entfilename = NULL;
930         }
        }

        return entnode;
    }

    /*
     * Record the addition of a new subdirectory DIR in PARENT.  PARENT
     * may be NULL, which means the current directory.  ENTRIES is the
     * current entries list; it may be NULL, which means that it need not
940  * be updated.
     */

    void
    Subdir_Register (entries, parent, dir)
         List *entries;
         const char *parent;
         const char *dir;
    {
        Entnode *entnode;
950

        /* Ignore attempts to register ".".  These can happen in the
           server code.  */
        if (dir[0] == '.' && dir[1] == '\0')
            return;

        entnode = subdir_record ('A', parent, dir);

        if (entries != NULL && (parent == NULL || strcmp (parent, ".") == 0))
            (void) AddEntryNode (entries, entnode);
960     else
            Entnode_Destroy (entnode);
    }

    /*
     * Record the removal of a subdirectory.  The arguments are the same
     * as for Subdir_Register.
     */

    void
970 Subdir_Deregister (entries, parent, dir)
         List *entries;
         const char *parent;
         const char *dir;
    {
        Entnode *entnode;

        entnode = subdir_record ('R', parent, dir);
        Entnode_Destroy (entnode);

980     if (entries != NULL && (parent == NULL || strcmp (parent, ".") == 0))
        {
            Node *p;

            p = findnode_fn (entries, dir);
            if (p != NULL)
                delnode (p);
        }
    }
```

```
990

    /* OK, the following base_* code tracks the revisions of the files in
       CVS/Base.  We do this in a file CVS/Baserev.  Separate from
       CVS/Entries because it needs to go in separate data structures
       anyway (the name in Entries must be unique), so this seemed
       cleaner.  The business of rewriting the whole file in
       base_deregister and base_register is the kind of thing we used to
       do for Entries and which turned out to be slow, which is why there
       is now the Entries.Log machinery.  So maybe from that point of
1000   view it is a mistake to do this separately from Entries, I dunno.

       We also need something analogous for:

       1. CVS/Template (so we can update the Template file automagically
       without the user needing to check out a new working directory).
       Updating would probably print a message (that part might be
       optional, although probably it should be visible because not all
       cvs commands would make the update happen and so it is a
       user-visible behavior).  Constructing version number for template
1010   is a bit hairy (base it on the timestamp on the server?  Or see if
       the template is in checkoutlist and if yes use its versioning and
       if no don't version it?). . . .

       2.  cvsignore (need to keep a copy in the working directory to do
       "cvs release" on the client side; see comment at src/release.c
       (release).  Would also allow us to stop needing Questionable.  */

    enum base_walk {
        /* Set the revision for FILE to *REV.  */
1020    BASE_REGISTER,
        /* Get the revision for FILE and put it in a newly malloc'd string
           in *REV, or put NULL if not mentioned.  */
        BASE_GET,
        /* Remove FILE.  */
        BASE_DEREGISTER
    };

    static void base_walk PROTO ((enum base_walk, struct file_info *, char **));

1030 /* Read through the lines in CVS/Baserev, taking the actions as documented
       for CODE.  */

    static void
    base_walk (code, finfo, rev)
        enum base_walk code;
        struct file_info *finfo;
        char **rev;
    {
        FILE *fp;
1040    char *line;
        size_t line_allocated;
        FILE *newf;
        char *baserev_fullname;
        char *baserevtmp_fullname;

        line = NULL;
        line_allocated = 0;
        newf = NULL;

1050    /* First compute the fullnames for the error messages.  This
           computation probably should be broken out into a separate function,
           as recurse.c does it too and places like Entries_Open should be
           doing it.  */
        baserev_fullname = xmalloc (sizeof (CVSADM_BASEREV)
                                    + strlen (finfo->update_dir)
                                    + 2);
        baserev_fullname[0] = '\0';
        baserevtmp_fullname = xmalloc (sizeof (CVSADM_BASEREVTMP)
                                       + strlen (finfo->update_dir)
1060                                   + 2);
        baserevtmp_fullname[0] = '\0';
        if (finfo->update_dir[0] != '\0')
        {
            strcat (baserev_fullname, finfo->update_dir);
            strcat (baserev_fullname, "/");
            strcat (baserevtmp_fullname, finfo->update_dir);
            strcat (baserevtmp_fullname, "/");
        }
        strcat (baserev_fullname, CVSADM_BASEREV);
1070    strcat (baserevtmp_fullname, CVSADM_BASEREVTMP);

        fp = CVS_FOPEN (CVSADM_BASEREV, "r");
        if (fp == NULL)
        {
            if (!existence_error (errno))
            {
                error (0, errno, "cannot open %s for reading", baserev_fullname);
                goto out;
```

```
        }
    }

    switch (code)
    {
        case BASE_REGISTER:
        case BASE_DEREGISTER:
            newf = CVS_FOPEN (CVSADM_BASEREVTMP, "w");
            if (newf == NULL)
            {
                error (0, errno, "cannot open %s for writing",
                       baserevtmp_fullname);
                goto out;
            }
            break;
        case BASE_GET:
            *rev = NULL;
            break;
    }

    if (fp != NULL)
    {
        while (getline (&line, &line_allocated, fp) >= 0)
        {
            char *linefile;
            char *p;
            char *linerev;

            if (line[0] != 'B')
                /* Ignore, for future expansion.  */
                continue;

            linefile = line + 1;
            p = strchr (linefile, '/');
            if (p == NULL)
                /* Syntax error, ignore.  */
                continue;
            linerev = p + 1;
            p = strchr (linerev, '/');
            if (p == NULL)
                continue;

            linerev[-1] = '\0';
            if (fncmp (linefile, finfo->file) == 0)
            {
                switch (code)
                {
                case BASE_REGISTER:
                case BASE_DEREGISTER:
                    /* Don't copy over the old entry, we don't want it.  */
                    break;
                case BASE_GET:
                    *p = '\0';
                    *rev = xstrdup (linerev);
                    *p = '/';
                    goto got_it;
                }
            }
            else
            {
                linerev[-1] = '/';
                switch (code)
                {
                case BASE_REGISTER:
                case BASE_DEREGISTER:
                    if (fprintf (newf, "%s\n", line) < 0)
                        error (0, errno, "error writing %s",
                               baserevtmp_fullname);
                    break;
                case BASE_GET:
                    break;
                }
            }
        }
        if (ferror (fp))
            error (0, errno, "cannot read %s", baserev_fullname);
    }
got_it:

    if (code == BASE_REGISTER)
    {
        if (fprintf (newf, "B%s/%s/\n", finfo->file, *rev) < 0)
            error (0, errno, "error writing %s",
                   baserevtmp_fullname);
    }

out:

    if (line != NULL)
        free (line);
```

```
    if (fp != NULL)
    {
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", baserev_fullname);
    }
    if (newf != NULL)
    {
        if (fclose (newf) < 0)
            error (0, errno, "cannot close %s", baserevtmp_fullname);
        rename_file (CVSADM_BASEREVTMP, CVSADM_BASEREV);
    }

    free (baserev_fullname);
    free (baserevtmp_fullname);
}

/* Return, in a newly malloc'd string, the revision for FILE in CVS/Baserev,
   or NULL if not listed.  */

char *
base_get (finfo)
    struct file_info *finfo;
{
    char *rev;
    base_walk (BASE_GET, finfo, &rev);
    return rev;
}

/* Set the revision for FILE to REV.  */

void
base_register (finfo, rev)
    struct file_info *finfo;
    char *rev;
{
    base_walk (BASE_REGISTER, finfo, &rev);
}

/* Remove FILE.  */

void
base_deregister (finfo)
    struct file_info *finfo;
{
    base_walk (BASE_DEREGISTER, finfo, NULL);
}
```

A.18 error.c

```
/* error.c - error handler for noninteractive utilities
   Copyright (C) 1990-1992 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

/* David MacKenzie */
/* Brian Berliner added support for CVS */

#include "cvs.h"

#include <stdio.h>

/* If non-zero, error will use the CVS protocol to stdout to report error
   messages.  This will only be set in the CVS server parent process;
   most other code is run via do_cvs_command, which forks off a child
   process and packages up its stderr in the protocol.  */
int error_use_protocol;

#ifdef HAVE_VPRINTF

#ifdef __STDC__
#include <stdarg.h>
#define VA_START(args, lastarg) va_start(args, lastarg)
#else /* ! __STDC__ */
#include <varargs.h>
#define VA_START(args, lastarg) va_start(args)
#endif /* __STDC__ */

#else /* ! HAVE_VPRINTF */

#ifdef HAVE_DOPRNT
#define va_alist args
#define va_dcl int args;
#else /* ! HAVE_DOPRNT */
#define va_alist a1, a2, a3, a4, a5, a6, a7, a8
#define va_dcl char *a1, *a2, *a3, *a4, *a5, *a6, *a7, *a8;
#endif /* HAVE_DOPRNT */

#endif /* HAVE_VPRINTF */

#if STDC_HEADERS
#include <stdlib.h>
#include <string.h>
#else /* ! STDC_HEADERS */
#ifdef __STDC__
void exit(int status);
#else /* ! __STDC__ */
void exit ();
#endif /* __STDC__ */
#endif /* STDC_HEADERS */

#ifndef strerror
extern char *strerror ();
#endif

extern int vasprintf ();

void
error_exit PROTO ((void))
{
    Lock_Cleanup();
#ifdef SERVER_SUPPORT
    if (server_active)
        server_cleanup (0);
#endif
#ifdef SYSTEM_CLEANUP
    /* Hook for OS-specific behavior, for example socket subsystems on
       NT and OS2 or dealing with windows and arguments on Mac.  */
    SYSTEM_CLEANUP ();
#endif
    exit (EXIT_FAILURE);
}

/* Print the program name and error message MESSAGE, which is a printf-style
   format string with optional args.
   If ERRNUM is nonzero, print its corresponding system error message.
   Exit with status EXIT_FAILURE if STATUS is nonzero.  If MESSAGE is "",
   no need to print a message.

   I think this is largely cleaned up to the point where it does the right
```

```
    thing for the server, whether the normal server_active (child process)
    case or the error_use_protocol (parent process) case.  The one exception
    is that STATUS nonzero for error_use_protocol probably doesn't work yet;
    in that case still need to use the pending_error machinery in server.c.

    error() does not molest errno; some code (e.g. Entries_Open) depends
    on being able to say something like:
      error (0, 0, "foo");
      error (0, errno, "bar");

    */

/* VARARGS */
void
#if defined (HAVE_VPRINTF) && defined (__STDC__)
error (int status, int errnum, const char *message, ...)
#else
error (status, errnum, message, va_alist)
    int status;
    int errnum;
    const char *message;
    va_dcl
#endif
{
    /* Prevent strtoul (via int_vasprintf) from clobbering it.  */
    int save_errno = errno;

#ifdef HAVE_VPRINTF
    if (message[0] != '\0')
    {
        va_list args;
        char *mess = NULL;
        char *entire;
        size_t len;

        VA_START (args, message);
        vasprintf (&mess, message, args);
        va_end (args);

        if (mess == NULL)
        {
            entire = NULL;
            status = 1;
        }
        else
        {
            len = strlen (mess) + strlen (program_name) + 80;
            if (command_name != NULL)
                len += strlen (command_name);
            if (errnum != 0)
                len += strlen (strerror (errnum));
            entire = malloc (len);
            if (entire == NULL)
            {
                free (mess);
                status = 1;
            }
            else
            {
                strcpy (entire, program_name);
                if (command_name != NULL && command_name[0] != '\0')
                {
                    strcat (entire, " ");
                    if (status != 0)
                        strcat (entire, "[");
                    strcat (entire, command_name);
                    if (status != 0)
                        strcat (entire, " aborted]");
                }
                strcat (entire, ": ");
                strcat (entire, mess);
                if (errnum != 0)
                {
                    strcat (entire, ": ");
                    strcat (entire, strerror (errnum));
                }
                strcat (entire, "\n");
                free (mess);
            }
        }
        cvs_outerr (entire ? entire : "out of memory\n", 0);
        if (entire != NULL)
            free (entire);
    }

#else /* No HAVE_VPRINTF */
    /* I think that all relevant systems have vprintf these days.  But
       just in case, I'm leaving this code here.  */

    if (message[0] != '\0')
```

```
    {
180     FILE *out = stderr;

        if (error_use_protocol)
        {
            out = stdout;
            printf ("E ");
        }

        if (command_name && *command_name)
        {
190         if (status)
                fprintf (out, "%s [%s aborted]: ", program_name, command_name);
            else
                fprintf (out, "%s %s: ", program_name, command_name);
        }
        else
            fprintf (out, "%s: ", program_name);

#ifdef HAVE_VPRINTF
        VA_START (args, message);
200     vfprintf (out, message, args);
        va_end (args);
#else
#ifdef HAVE_DOPRNT
        _doprnt (message, &args, out);
#else
        fprintf (out, message, a1, a2, a3, a4, a5, a6, a7, a8);
#endif
#endif
        if (errnum)
210         fprintf (out, ": %s", strerror (errnum));
        putc ('\n', out);

        /* In the error_use_protocol case, this probably does
           something useful.  In most other cases, I suspect it is a
           noop (either stderr is line buffered or we haven't written
           anything to stderr) or unnecessary (if stderr is not line
           buffered, maybe there is a reason. . . .).  */
        fflush (out);
    }
220
#endif /* No HAVE_VPRINTF */

    if (status)
        error_exit ();
    errno = save_errno;
}

/* Print the program name and error message MESSAGE, which is a printf-style
   format string with optional args to the file specified by FP.
230  If ERRNUM is nonzero, print its corresponding system error message.
   Exit with status EXIT_FAILURE if STATUS is nonzero.  */
/* VARARGS */
void
#if defined (HAVE_VPRINTF) && defined (__STDC__)
fperror (FILE *fp, int status, int errnum, char *message, ...)
#else
fperror (fp, status, errnum, message, va_alist)
    FILE *fp;
    int status;
240 int errnum;
    char *message;
    va_dcl
#endif
{
#ifdef HAVE_VPRINTF
    va_list args;
#endif

    fprintf (fp, "%s: ", program_name);
250 #ifdef HAVE_VPRINTF
    VA_START (args, message);
    vfprintf (fp, message, args);
    va_end (args);
#else
#ifdef HAVE_DOPRNT
    _doprnt (message, &args, fp);
#else
    fprintf (fp, message, a1, a2, a3, a4, a5, a6, a7, a8);
#endif
260 #endif
    if (errnum)
        fprintf (fp, ": %s", strerror (errnum));
    putc ('\n', fp);
    fflush (fp);
    if (status)
        error_exit ();
}
```

A.19 error.h

```
/* error.h - declaration for error-reporting function
   Copyright (C) 1995 Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#ifndef _error_h_
#define _error_h_

/* Add prototype support.  Normally this is done in cvs.h, but that
   doesn't get included from lib/savecwd.c.  */
#ifndef PROTO
#if defined (USE_PROTOTYPES) ? USE_PROTOTYPES : defined (__STDC__)
#define PROTO(ARGS) ARGS
#else
#define PROTO(ARGS) ()
#endif
#endif

#ifndef __attribute__
/* This feature is available in gcc versions 2.5 and later.  */
# if __GNUC__ < 2 || (__GNUC__ == 2 && __GNUC_MINOR__ < 5) || __STRICT_ANSI__
#  define __attribute__(Spec) /* empty */
# endif
/* The __-protected variants of 'format' and 'printf' attributes
   are accepted by gcc versions 2.6.4 (effectively 2.7) and later.  */
# if __GNUC__ < 2 || (__GNUC__ == 2 && __GNUC_MINOR__ < 7)
#  define __format__ format
#  define __printf__ printf
# endif
#endif

#ifdef __STDC__
void error (int, int, const char *, ...) \
  __attribute__ ((__format__ (__printf__, 3, 4)));
#else
void error ();
#endif

/* Exit due to an error.  Similar to error (1, 0, "message"), but call
   it in the case where the message has already been printed.  */
extern void error_exit PROTO ((void));

/* If non-zero, error will use the CVS protocol to report error
   messages.  This will only be set in the CVS server parent process;
   most other code is run via do_cvs_command, which forks off a child
   process and packages up its stderr in the protocol.  */
extern int error_use_protocol;

#endif /* _error_h_ */
```

A.20 expand_path.c

```
/* expand_path.c - expand environmental variables in passed in string
 *
 * The main routine is expand_path(), it is the routine that handles
 * the '~' character in four forms:
 *      ~name
 *      ~name/
 *      ~/
 *      ~
 * and handles environment variables contained within the pathname
 * which are defined by:
 *      ${var_name}   (var_name is the name of the environ variable)
 *      $var_name     (var_name ends w/ non-alphanumeric char other than '_')
 */

#include "cvs.h"
#include <sys/types.h>

static char *expand_variable PROTO((char *env, char *file, int line));


/* User variables.  */

List *variable_list = NULL;

static void variable_delproc PROTO ((Node *));

static void
variable_delproc (node)
    Node *node;
{
    free (node->data);
}

/* Currently used by -s option; we might want a way to set user
   variables in a file in the $CVSROOT/CVSROOT directory too.  */

void
variable_set (nameval)
    char *nameval;
{
    char *p;
    char *name;
    Node *node;

    p = nameval;
    while (isalnum (*p) || *p == '_')
        ++p;
    if (*p != '=')
        error (1, 0, "illegal character in user variable name in %s", nameval);
    if (p == nameval)
        error (1, 0, "empty user variable name in %s", nameval);
    name = xmalloc (p - nameval + 1);
    strncpy (name, nameval, p - nameval);
    name[p - nameval] = '\0';
    /* Make p point to the value.  */
    ++p;
    if (strchr (p, '\012') != NULL)
        error (1, 0, "linefeed in user variable value in %s", nameval);

    if (variable_list == NULL)
        variable_list = getlist ();

    node = findnode (variable_list, name);
    if (node == NULL)
    {
        node = getnode ();
        node->type = VARIABLE;
        node->delproc = variable_delproc;
        node->key = name;
        node->data = xstrdup (p);
        (void) addnode (variable_list, node);
    }
    else
    {
        /* Replace the old value.  For example, this means that -s
           options on the command line override ones from .cvsrc.  */
        free (node->data);
        node->data = xstrdup (p);
        free (name);
    }
}

/* This routine will expand the pathname to account for ~ and $
   characters as described above.  Returns a pointer to a newly
   malloc'd string.  If an error occurs, an error message is printed
   via error() and NULL is returned.  FILE and LINE are the filename
   and linenumber to include in the error message.  FILE must point
   to something; LINE can be zero to indicate the line number is not
```

```
   known.  */
90 char *
expand_path (name, file, line)
    char *name;
    char *file;
    int line;
{
    char *s;
    char *d;

    char *mybuf = NULL;
100 size_t mybuf_size = 0;
    char *buf = NULL;
    size_t buf_size = 0;

    size_t doff;

    char *result;

    /* Sorry this routine is so ugly; it is a head-on collision
       between the 'traditional' unix *d++ style and the need to
110    dynamically allocate.  It would be much cleaner (and probably
       faster, not that this is a bottleneck for CVS) with more use of
       strcpy & friends, but I haven't taken the effort to rewrite it
       thusly.  */

    /* First copy from NAME to MYBUF, expanding $<foo> as we go.  */
    s = name;
    d = mybuf;
    doff = d - mybuf;
    expand_string (&mybuf, &mybuf_size, doff + 1);
120 d = mybuf + doff;
    while ((*d++ = *s))
    {
        if (*s++ == '$')
        {
            char *p = d;
            char *e;
            int flag = (*s == '{');

            doff = d - mybuf;
130         expand_string (&mybuf, &mybuf_size, doff + 1);
            d = mybuf + doff;
            for (; (*d++ = *s); s++)
            {
                if (flag
                    ? *s =='}'
                    : isalnum (*s) == 0 && *s != '_')
                    break;
                doff = d - mybuf;
                expand_string (&mybuf, &mybuf_size, doff + 1);
140             d = mybuf + doff;
            }
            *--d = '\0';
            e = expand_variable (&p[flag], file, line);

            if (e)
            {
                doff = d - mybuf;
                expand_string (&mybuf, &mybuf_size, doff + 1);
                d = mybuf + doff;
150             for (d = &p[-1]; (*d++ = *e++);)
                {
                    doff = d - mybuf;
                    expand_string (&mybuf, &mybuf_size, doff + 1);
                    d = mybuf + doff;
                }
                --d;
                if (flag && *s)
                    s++;
            }
160         else
                /* expand_variable has already printed an error message.  */
                goto error_exit;
        }
        doff = d - mybuf;
        expand_string (&mybuf, &mybuf_size, doff + 1);
        d = mybuf + doff;
    }
    doff = d - mybuf;
    expand_string (&mybuf, &mybuf_size, doff + 1);
170 d = mybuf + doff;
    *d = '\0';

    /* Then copy from MYBUF to BUF, expanding ~.  */
    s = mybuf;
    d = buf;
    /* If you don't want ~username ~/ to be expanded simply remove
     * This entire if statement including the else portion
     */
```

```
    if (*s++ == '~')
180 {
        char *t;
        char *p=s;
        if (*s=='/' || *s==0)
            t = get_homedir ();
        else
        {
#ifdef GETPWNAM_MISSING
            for (; *p!='/' && *p; p++)
                ;
190         *p = 0;
            if (line != 0)
                error (0, 0,
                       "%s:%d:tilde expansion not supported on this system",
                       file, line);
            else
                error (0, 0, "%s:tilde expansion not supported on this system",
                       file);
            return NULL;
#else
200         struct passwd *ps;
            for (; *p!='/' && *p; p++)
                ;
            *p = 0;
            ps = getpwnam (s);
            if (ps == 0)
            {
                if (line != 0)
                    error (0, 0, "%s:%d: no such user %s",
                           file, line, s);
210             else
                    error (0, 0, "%s: no such user %s", file, s);
                return NULL;
            }
            t = ps->pw_dir;
#endif
        }
        doff = d - buf;
        expand_string (&buf, &buf_size, doff + 1);
        d = buf + doff;
220     while ((*d++ = *t++))
        {
            doff = d - buf;
            expand_string (&buf, &buf_size, doff + 1);
            d = buf + doff;
        }
        --d;
        if (*p == 0)
            *p = '/';       /* always add / */
        s=p;
230 }
    else
        --s;
    /* Kill up to here */
    doff = d - buf;
    expand_string (&buf, &buf_size, doff + 1);
    d = buf + doff;
    while ((*d++ = *s++))
    {
        doff = d - buf;
240     expand_string (&buf, &buf_size, doff + 1);
        d = buf + doff;
    }
    doff = d - buf;
    expand_string (&buf, &buf_size, doff + 1);
    d = buf + doff;
    *d = '\0';

    /* OK, buf contains the value we want to return.  Clean up and return
       it.  */
250 free (mybuf);
    /* Save a little memory with xstrdup; buf will tend to allocate
       more than it needs to.  */
    result = xstrdup (buf);
    free (buf);
    return result;

 error_exit:
    if (mybuf != NULL)
        free (mybuf);
260 if (buf != NULL)
        free (buf);
    return NULL;
}

static char *
expand_variable (name, file, line)
    char *name;
    char *file;
```

```
    int line;
{
    if (strcmp (name, CVSROOT_ENV) == 0)
        return CVSroot_original;
    else if (strcmp (name, "RCSBIN") == 0)
    {
        error (0, 0, "RCSBIN internal variable is no longer supported");
        return NULL;
    }
    else if (strcmp (name, EDITOR1_ENV) == 0)
        return Editor;
    else if (strcmp (name, EDITOR2_ENV) == 0)
        return Editor;
    else if (strcmp (name, EDITOR3_ENV) == 0)
        return Editor;
    else if (strcmp (name, "USER") == 0)
        return getcaller ();
    else if (isalpha (name[0]))
    {
        /* These names are reserved for future versions of CVS,
           so that is why it is an error.  */
        if (line != 0)
            error (0, 0, "%s:%d: no such internal variable $%s",
                   file, line, name);
        else
            error (0, 0, "%s: no such internal variable $%s",
                   file, name);
        return NULL;
    }
    else if (name[0] == '=')
    {
        Node *node;
        /* Crazy syntax for a user variable.  But we want
           *something* that lets the user name a user variable
           anything he wants, without interference from
           (existing or future) internal variables.  */
        node = findnode (variable_list, name + 1);
        if (node == NULL)
        {
            if (line != 0)
                error (0, 0, "%s:%d: no such user variable ${%s}",
                       file, line, name);
            else
                error (0, 0, "%s: no such user variable ${%s}",
                       file, name);
            return NULL;
        }
        return node->data;
    }
    else
    {
        /* It is an unrecognized character.  We return an error to
           reserve these for future versions of CVS; it is plausible
           that various crazy syntaxes might be invented for inserting
           information about revisions, branches, etc.  */
        if (line != 0)
            error (0, 0, "%s:%d: unrecognized variable syntax %s",
                   file, line, name);
        else
            error (0, 0, "%s: unrecognized variable syntax %s",
                   file, name);
        return NULL;
    }
}
```

A.21 fileattr.c

```
/* Implementation for file attribute munging features.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include "cvs.h"
#include "getline.h"
#include "fileattr.h"
#include <assert.h>

static void fileattr_read PROTO ((void));
static int writeattr_proc PROTO ((Node *, void *));

/* Where to look for CVSREP_FILEATTR.  */
static char *fileattr_stored_repos;

/* The in-memory attributes.  */
static List *attrlist;
static char *fileattr_default_attrs;
/* We have already tried to read attributes and failed in this directory
   (for example, there is no CVSREP_FILEATTR file).  */
static int attr_read_attempted;

/* Have the in-memory attributes been modified since we read them?  */
static int attrs_modified;

/* More in-memory attributes: linked list of unrecognized
   fileattr lines.  We pass these on unchanged.  */
struct unrecog {
    char *line;
    struct unrecog *next;
};
static struct unrecog *unrecog_head;

/* Note that if noone calls fileattr_get, this is very cheap.  No stat(),
   no open(), no nothing.  */
void
fileattr_startdir (repos)
    char *repos;
{
    assert (fileattr_stored_repos == NULL);
    fileattr_stored_repos = xstrdup (repos);
    assert (attrlist == NULL);
    attr_read_attempted = 0;
    assert (unrecog_head == NULL);
}

static void
fileattr_delproc (node)
    Node *node;
{
    assert (node->data != NULL);
    free (node->data);
    node->data = NULL;
}

/* Read all the attributes for the current directory into memory.  */
static void
fileattr_read ()
{
    char *fname;
    FILE *fp;
    char *line = NULL;
    size_t line_len = 0;

    /* If there are no attributes, don't waste time repeatedly looking
       for the CVSREP_FILEATTR file.  */
    if (attr_read_attempted)
        return;

    /* If NULL was passed to fileattr_startdir, then it isn't kosher to look
       at attributes.  */
    assert (fileattr_stored_repos != NULL);

    fname = xmalloc (strlen (fileattr_stored_repos)
                     + 1
                     + sizeof (CVSREP_FILEATTR)
                     + 1);

    strcpy (fname, fileattr_stored_repos);
    strcat (fname, "/");
```

```
    strcat (fname, CVSREP_FILEATTR);

    attr_read_attempted = 1;
    fp = CVS_FOPEN (fname, FOPEN_BINARY_READ);
    if (fp == NULL)
    {
        if (!existence_error (errno))
            error (0, errno, "cannot read %s", fname);
        free (fname);
        return;
    }
    attrlist = getlist ();
    while (1) {
        int nread;
        nread = getline (&line, &line_len, fp);
        if (nread < 0)
            break;
        /* Remove trailing newline.  */
        line[nread - 1] = '\0';
        if (line[0] == 'F')
        {
            char *p;
            Node *newnode;

            p = strchr (line, '\t');
            if (p == NULL)
                error (1, 0,
                       "file attribute database corruption: tab missing in %s",
                       fname);
            *p++ = '\0';
            newnode = getnode ();
            newnode->type = FILEATTR;
            newnode->delproc = fileattr_delproc;
            newnode->key = xstrdup (line + 1);
            newnode->data = xstrdup (p);
            if (addnode (attrlist, newnode) != 0)
                /* If the same filename appears twice in the file, discard
                   any line other than the first for that filename.  This
                   is the way that CVS has behaved since file attributes
                   were first introduced.  */
                free (newnode);
        }
        else if (line[0] == 'D')
        {
            char *p;
            /* Currently nothing to skip here, but for future expansion,
               ignore anything located here.  */
            p = strchr (line, '\t');
            if (p == NULL)
                error (1, 0,
                       "file attribute database corruption: tab missing in %s",
                       fname);
            ++p;
            fileattr_default_attrs = xstrdup (p);
        }
        else
        {
            /* Unrecognized type, we want to just preserve the line without
               changing it, for future expansion.  */
            struct unrecog *new;

            new = (struct unrecog *) xmalloc (sizeof (struct unrecog));
            new->line = xstrdup (line);
            new->next = unrecog_head;
            unrecog_head = new;
        }
    }
    if (ferror (fp))
        error (0, errno, "cannot read %s", fname);
    if (line != NULL)
        free (line);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fname);
    attrs_modified = 0;
    free (fname);
}

char *
fileattr_get (filename, attrname)
    const char *filename;
    const char *attrname;
{
    Node *node;
    size_t attrname_len = strlen (attrname);
    char *p;

    if (attrlist == NULL)
        fileattr_read ();
    if (attrlist == NULL)
        /* Either nothing has any attributes, or fileattr_read already printed
```

```
            an error message.  */
180         return NULL;

        if (filename == NULL)
            p = fileattr_default_attrs;
        else
        {
            node = findnode (attrlist, filename);
            if (node == NULL)
                /* A file not mentioned has no attributes.  */
                return NULL;
190         p = node->data;
        }
        while (p)
        {
            if (strncmp (attrname, p, attrname_len) == 0
                && p[attrname_len] == '=')
            {
                /* Found it.  */
                return p + attrname_len + 1;
            }
200         p = strchr (p, ';');
            if (p == NULL)
                break;
            ++p;
        }
        /* The file doesn't have this attribute.  */
        return NULL;
    }

    char *
210 fileattr_get0 (filename, attrname)
        const char *filename;
        const char *attrname;
    {
        char *cp;
        char *cpend;
        char *retval;

        cp = fileattr_get (filename, attrname);
        if (cp == NULL)
220         return NULL;
        cpend = strchr (cp, ';');
        if (cpend == NULL)
            cpend = cp + strlen (cp);
        retval = xmalloc (cpend - cp + 1);
        strncpy (retval, cp, cpend - cp);
        retval[cpend - cp] = '\0';
        return retval;
    }

230 char *
    fileattr_modify (list, attrname, attrval, namevalsep, entsep)
        char *list;
        const char *attrname;
        const char *attrval;
        int namevalsep;
        int entsep;
    {
        char *retval;
        char *rp;
240     size_t attrname_len = strlen (attrname);

        /* Portion of list before the attribute to be replaced.  */
        char *pre;
        char *preend;
        /* Portion of list after the attribute to be replaced.  */
        char *post;

        char *p;
        char *p2;
250
        p = list;
        pre = list;
        preend = NULL;
        /* post is NULL unless set otherwise.  */
        post = NULL;
        p2 = NULL;
        if (list != NULL)
        {
            while (1) {
260             p2 = strchr (p, entsep);
                if (p2 == NULL)
                {
                    p2 = p + strlen (p);
                    if (preend == NULL)
                        preend = p2;
                }
                else
                    ++p2;
```

```
            if (strncmp (attrname, p, attrname_len) == 0
270             && p[attrname_len] == namevalsep)
            {
                /* Found it.  */
                preend = p;
                if (preend > list)
                    /* Don't include the preceding entsep.  */
                    --preend;

                post = p2;
            }
280         if (p2[0] == '\0')
                break;
            p = p2;
        }
    }
    if (post == NULL)
        post = p2;

    if (preend == pre && attrval == NULL && post == p2)
        return NULL;
290
    retval = xmalloc ((preend - pre)
                      + 1
                      + (attrval == NULL ? 0 : (attrname_len + 1
                                                + strlen (attrval)))
                      + 1
                      + (p2 - post)
                      + 1);
    if (preend != pre)
    {
300     strncpy (retval, pre, preend - pre);
        rp = retval + (preend - pre);
        if (attrval != NULL)
            *rp++ = entsep;
        *rp = '\0';
    }
    else
        retval[0] = '\0';
    if (attrval != NULL)
    {
310     strcat (retval, attrname);
        rp = retval + strlen (retval);
        *rp++ = namevalsep;
        strcpy (rp, attrval);
    }
    if (post != p2)
    {
        rp = retval + strlen (retval);
        if (preend != pre || attrval != NULL)
            *rp++ = entsep;
320     strncpy (rp, post, p2 - post);
        rp += p2 - post;
        *rp = '\0';
    }
    return retval;
}

void
fileattr_set (filename, attrname, attrval)
    const char *filename;
330 const char *attrname;
    const char *attrval;
{
    Node *node;
    char *p;

    if (filename == NULL)
    {
        p = fileattr_modify (fileattr_default_attrs, attrname, attrval,
                             '=', ';');
340     if (fileattr_default_attrs != NULL)
            free (fileattr_default_attrs);
        fileattr_default_attrs = p;
        attrs_modified = 1;
        return;
    }
    if (attrlist == NULL)
        fileattr_read ();
    if (attrlist == NULL)
    {
350     /* Not sure this is a graceful way to handle things
           in the case where fileattr_read was unable to read the file.  */
        /* No attributes existed previously.  */
        attrlist = getlist ();
    }

    node = findnode (attrlist, filename);
    if (node == NULL)
    {
```

```
        if (attrval == NULL)
            /* Attempt to remove an attribute which wasn't there.  */
            return;

        /* First attribute for this file.  */
        node = getnode ();
        node->type = FILEATTR;
        node->delproc = fileattr_delproc;
        node->key = xstrdup (filename);
        node->data = xmalloc (strlen (attrname) + 1 + strlen (attrval) + 1);
        strcpy (node->data, attrname);
        strcat (node->data, "=");
        strcat (node->data, attrval);
        addnode (attrlist, node);
    }

    p = fileattr_modify (node->data, attrname, attrval, '=', ';');
    if (p == NULL)
        delnode (node);
    else
    {
        free (node->data);
        node->data = p;
    }

    attrs_modified = 1;
}

char *
fileattr_getall (filename)
    const char *filename;
{
    Node *node;
    char *p;

    if (attrlist == NULL)
        fileattr_read ();
    if (attrlist == NULL)
        /* Either nothing has any attributes, or fileattr_read already printed
           an error message.  */
        return NULL;

    if (filename == NULL)
        p = fileattr_default_attrs;
    else
    {
        node = findnode (attrlist, filename);
        if (node == NULL)
            /* A file not mentioned has no attributes.  */
            return NULL;
        p = node->data;
    }
    return xstrdup (p);
}

void
fileattr_setall (filename, attrs)
    const char *filename;
    const char *attrs;
{
    Node *node;

    if (filename == NULL)
    {
        if (fileattr_default_attrs != NULL)
            free (fileattr_default_attrs);
        fileattr_default_attrs = xstrdup (attrs);
        attrs_modified = 1;
        return;
    }
    if (attrlist == NULL)
        fileattr_read ();
    if (attrlist == NULL)
    {
        /* Not sure this is a graceful way to handle things
           in the case where fileattr_read was unable to read the file.  */
        /* No attributes existed previously.  */
        attrlist = getlist ();
    }

    node = findnode (attrlist, filename);
    if (node == NULL)
    {
        /* The file had no attributes.  Add them if we have any to add.  */
        if (attrs != NULL)
        {
            node = getnode ();
            node->type = FILEATTR;
            node->delproc = fileattr_delproc;
            node->key = xstrdup (filename);
```

```
            node->data = xstrdup (attrs);
            addnode (attrlist, node);
        }
    }
    else
    {
        if (attrs == NULL)
            delnode (node);
        else
        {
            free (node->data);
            node->data = xstrdup (attrs);
        }
    }

    attrs_modified = 1;
}

void
fileattr_newfile (filename)
    const char *filename;
{
    Node *node;

    if (attrlist == NULL)
        fileattr_read ();

    if (fileattr_default_attrs == NULL)
        return;

    if (attrlist == NULL)
    {
        /* Not sure this is a graceful way to handle things
           in the case where fileattr_read was unable to read the file.  */
        /* No attributes existed previously.  */
        attrlist = getlist ();
    }

    node = getnode ();
    node->type = FILEATTR;
    node->delproc = fileattr_delproc;
    node->key = xstrdup (filename);
    node->data = xstrdup (fileattr_default_attrs);
    addnode (attrlist, node);
    attrs_modified = 1;
}

static int
writeattr_proc (node, data)
    Node *node;
    void *data;
{
    FILE *fp = (FILE *)data;
    fputs ("F", fp);
    fputs (node->key, fp);
    fputs ("\t", fp);
    fputs (node->data, fp);
    fputs ("\012", fp);
    return 0;
}

void
fileattr_write ()
{
    FILE *fp;
    char *fname;
    mode_t omask;

    if (!attrs_modified)
        return;

    if (noexec)
        return;

    /* If NULL was passed to fileattr_startdir, then it isn't kosher to set
       attributes.  */
    assert (fileattr_stored_repos != NULL);

    fname = xmalloc (strlen (fileattr_stored_repos)
                     + 1
                     + sizeof (CVSREP_FILEATTR)
                     + 1);

    strcpy (fname, fileattr_stored_repos);
    strcat (fname, "/");
    strcat (fname, CVSREP_FILEATTR);

    if (list_isempty (attrlist)
        && fileattr_default_attrs == NULL
        && unrecog_head == NULL)
```

```
{
    /* There are no attributes.  */
    if (unlink_file (fname) < 0)
    {
        if (!existence_error (errno))
        {
            error (0, errno, "cannot remove %s", fname);
        }
    }

    /* Now remove CVSREP directory, if empty.  The main reason we bother
       is that CVS 1.6 and earlier will choke if a CVSREP directory
       exists, so provide the user a graceful way to remove it.  */
    strcpy (fname, fileattr_stored_repos);
    strcat (fname, "/");
    strcat (fname, CVSREP);
    if (CVS_RMDIR (fname) < 0)
    {
        if (errno != ENOTEMPTY

            /* Don't know why we would be here if there is no CVSREP
               directory, but it seemed to be happening anyway, so
               check for it.  */
            && !existence_error (errno))
            error (0, errno, "cannot remove %s", fname);
    }

    free (fname);
    return;
}

omask = umask (cvsumask);
fp = CVS_FOPEN (fname, FOPEN_BINARY_WRITE);
if (fp == NULL)
{
    if (existence_error (errno))
    {
        /* Maybe the CVSREP directory doesn't exist.  Try creating it.  */
        char *repname;

        repname = xmalloc (strlen (fileattr_stored_repos)
                           + 1
                           + sizeof (CVSREP)
                           + 1);
        strcpy (repname, fileattr_stored_repos);
        strcat (repname, "/");
        strcat (repname, CVSREP);

        if (CVS_MKDIR (repname, 0777) < 0 && errno != EEXIST)
        {
            error (0, errno, "cannot make directory %s", repname);
            (void) umask (omask);
            free (repname);
            return;
        }
        free (repname);

        fp = CVS_FOPEN (fname, FOPEN_BINARY_WRITE);
    }
    if (fp == NULL)
    {
        error (0, errno, "cannot write %s", fname);
        (void) umask (omask);
        return;
    }
}
(void) umask (omask);

/* First write the "F" attributes.  */
walklist (attrlist, writeattr_proc, fp);

/* Then the "D" attribute.  */
if (fileattr_default_attrs != NULL)
{
    fputs ("D\t", fp);
    fputs (fileattr_default_attrs, fp);
    fputs ("\012", fp);
}

/* Then any other attributes.  */
while (unrecog_head != NULL)
{
    struct unrecog *p;

    p = unrecog_head;
    fputs (p->line, fp);
    fputs ("\012", fp);

    unrecog_head = p->next;
    free (p->line);
```

```
        free (p);
    }

    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", fname);
    attrs_modified = 0;
    free (fname);
}

void
fileattr_free ()
{
    /* Note that attrs_modified will ordinarily be zero, but there are
       a few cases in which fileattr_write will fail to zero it (if
       noexec is set, or error conditions).  This probably is the way
       it should be.  */
    dellist (&attrlist);
    if (fileattr_stored_repos != NULL)
        free (fileattr_stored_repos);
    fileattr_stored_repos = NULL;
    if (fileattr_default_attrs != NULL)
        free (fileattr_default_attrs);
    fileattr_default_attrs = NULL;
}
```

A.22 fileattr.h

```
/* Declarations for file attribute munging features.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#ifndef FILEATTR_H

/* File containing per-file attributes.  Format is a series of entries:

   ENT-TYPE FILENAME <tab> ATTRNAME = ATTRVAL
     {; ATTRNAME = ATTRVAL} <linefeed>

   ENT-TYPE is 'F' for a file, in which case the entry specifies the
   attributes for that file.

   ENT-TYPE is 'D', and FILENAME empty, to specify default attributes
   to be used for newly added files.

   Other ENT-TYPE are reserved for future expansion.  CVS 1.9 and older
   will delete them any time it writes file attributes.  Current versions
   of CVS will preserve them.

   Note that the order of the line is not significant; CVS is free to
   rearrange them at its convenience.

   There is currently no way of quoting tabs or linefeeds in the
   filename, '=' in ATTRNAME, ';' in ATTRVAL, etc.  I'm not sure
   whether I think we need one.  Note: the current implementation also
   doesn't handle '\0' in any of the fields.

   By convention, ATTRNAME starting with '_' is for an attribute given
   special meaning by CVS; other ATTRNAMEs are for user-defined attributes
   (or will be, once we add commands to manipulate user-defined attributes).

   Builtin attributes:

   _watched: Present means the file is watched and should be checked out
   read-only.

   _watchers: Users with watches for this file.  Value is
   WATCHER > TYPE { , WATCHER > TYPE }
   where WATCHER is a username, and TYPE is edit,unedit,commit separated by
   + (or nothing if none; there is no "none" or "all" keyword).

   _editors: Users editing this file.  Value is
   EDITOR > VAL { , EDITOR > VAL }
   where EDITOR is a username, and VAL is TIME+HOSTNAME+PATHNAME, where
   TIME is when the "cvs edit" command happened,
   and HOSTNAME and PATHNAME are for the working directory.  */

#define CVSREP_FILEATTR "CVS/fileattr"

/* Prepare for a new directory with repository REPOS.  If REPOS is NULL,
   then prepare for a "non-directory"; the caller can call fileattr_write
   and fileattr_free, but must not call fileattr_get or fileattr_set.  */
extern void fileattr_startdir PROTO ((char *repos));

/* Get the attribute ATTRNAME for file FILENAME.  The return value
   points into memory managed by the fileattr_* routines, should not
   be altered by the caller, and is only good until the next call to
   fileattr_clear or fileattr_set.  It points to the value, terminated
   by '\0' or ';'.  Return NULL if said file lacks said attribute.
   If FILENAME is NULL, return default attributes (attributes for
   files created in the future).  */
extern char *fileattr_get PROTO ((const char *filename, const char *attrname));

/* Like fileattr_get, but return a pointer to a newly malloc'd string
   terminated by '\0' (or NULL if said file lacks said attribute).  */
extern char *fileattr_get0 PROTO ((const char *filename,
                                   const char *attrname));

/* This is just a string manipulation function; it does not manipulate
   file attributes as such.

   LIST is in the format

   ATTRNAME NAMEVALSEP ATTRVAL {ENTSEP ATTRNAME NAMEVALSEP ATTRVAL}

   And we want to put in an attribute with name NAME and value VAL,
   replacing the already-present attribute with name NAME if there is
   one.  Or if VAL is NULL remove attribute NAME.  Return a new
```

```
   malloc'd list; don't muck with the one passed in.  If we are removing
   the last attribute return NULL.  LIST can be NULL to mean that we
   started out without any attributes.

   Examples:

   fileattr_modify ("abc=def", "xxx", "val", '=', ';')) => "abc=def;xxx=val"
   fileattr_modify ("abc=def", "abc", "val", '=', ';')) => "abc=val"
   fileattr_modify ("abc=v1;def=v2", "abc", "val", '=', ';'))
     => "abc=val;def=v2"
   fileattr_modify ("abc=v1;def=v2", "def", "val", '=', ';'))
     => "abc=v1;def=val"
   fileattr_modify ("abc=v1;def=v2", "xxx", "val", '=', ';'))
     => "abc=v1;def=v2;xxx=val"
   fileattr_modify ("abc=v1;def=v2;ghi=v3", "def", "val", '=', ';'))
     => "abc=v1;def=val;ghi=v3"
*/

extern char *fileattr_modify PROTO ((char *list, const char *attrname,
                                     const char *attrval, int namevalsep,
                                     int entsep));

/* Set attribute ATTRNAME for file FILENAME to ATTRVAL.  If ATTRVAL is NULL,
   the attribute is removed.  Changes are not written to disk until the
   next call to fileattr_write.  If FILENAME is NULL, set attributes for
   files created in the future.  If ATTRVAL is NULL, remove that attribute.  */
extern void fileattr_set PROTO ((const char *filename, const char *attrname,
                                 const char *attrval));

/* Get all the attributes for file FILENAME.  They are returned as malloc'd
   data in an unspecified format which is guaranteed only to be good for
   passing to fileattr_setall, or NULL if no attributes.  If FILENAME is
   NULL, get default attributes.  */
extern char *fileattr_getall PROTO ((const char *filename));

/* Set the attributes for file FILENAME to ATTRS, overwriting all previous
   attributes for that file.  ATTRS was obtained from a previous call to
   fileattr_getall (malloc'd data or NULL).  */
extern void fileattr_setall PROTO ((const char *filename, const char *attrs));

/* Set the attributes for file FILENAME in whatever manner is appropriate
   for a newly created file.  */
extern void fileattr_newfile PROTO ((const char *filename));

/* Write out all modified attributes.  */
extern void fileattr_write PROTO ((void));

/* Free all memory allocated by fileattr_*.  */
extern void fileattr_free PROTO ((void));

#define FILEATTR_H 1
#endif /* fileattr.h */
```

A.23 filesubr.c

```
/* filesubr.c — subroutines for dealing with files
   Jim Blandy <jimbcyclic.com>

   This file is part of GNU CVS.

   GNU CVS is free software; you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published by the
   Free Software Foundation; either version 2, or (at your option) any
   later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*/

/* These functions were moved out of subr.c because they need different
   definitions under operating systems (like, say, Windows NT) with different
   file system semantics.  */

#include "cvs.h"

static int deep_remove_dir PROTO((const char *path));

/*
 * Copies "from" to "to".
 */
void
copy_file (from, to)
    const char *from;
    const char *to;
{
    struct stat sb;
    struct utimbuf t;
    int fdin, fdout;

    if (trace)
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> copy(%s,%s)\n",
                        (server_active) ? 'S' : ' ', from, to);
#else
        (void) fprintf (stderr, "-> copy(%s,%s)\n", from, to);
#endif
    if (noexec)
        return;

    /* If the file to be copied is a link or a device, then just create
       the new link or device appropriately. */
    if (islink (from))
    {
        char *source = xreadlink (from);
        symlink (source, to);
        free (source);
        return;
    }

    if (isdevice (from))
    {
        if (stat (from, &sb) < 0)
            error (1, errno, "cannot stat %s", from);
        mknod (to, sb.st_mode, sb.st_rdev);
    }
    else
    {
        /* Not a link or a device... probably a regular file. */
        if ((fdin = open (from, O_RDONLY)) < 0)
            error (1, errno, "cannot open %s for copying", from);
        if (fstat (fdin, &sb) < 0)
            error (1, errno, "cannot fstat %s", from);
        if ((fdout = creat (to, (int) sb.st_mode & 07777)) < 0)
            error (1, errno, "cannot create %s for copying", to);
        if (sb.st_size > 0)
        {
            char buf[BUFSIZ];
            int n;

            for (;;)
            {
                n = read (fdin, buf, sizeof(buf));
                if (n == -1)
                {
#ifdef EINTR
                    if (errno == EINTR)
                        continue;
#endif
                    error (1, errno, "cannot read file %s for copying", from);
                }
                else if (n == 0)
                    break;
```

```
90              if (write(fdout, buf, n) != n) {
                    error (1, errno, "cannot write file %s for copying", to);
                }
            }

#ifdef HAVE_FSYNC
            if (fsync (fdout))
                error (1, errno, "cannot fsync file %s after copying", to);
#endif
        }
100
        if (close (fdin) < 0)
            error (0, errno, "cannot close %s", from);
        if (close (fdout) < 0)
            error (1, errno, "cannot close %s", to);
    }

    /* now, set the times for the copied file to match those of the original */
    memset ((char *) &t, 0, sizeof (t));
    t.actime = sb.st_atime;
110 t.modtime = sb.st_mtime;
    (void) utime (to, &t);
}

/* FIXME-krp: these functions would benefit from caching the char * &
   stat buf.  */

/*
 * Returns non-zero if the argument file is a directory, or is a symbolic
 * link which points to a directory.
120 */
int
isdir (file)
    const char *file;
{
    struct stat sb;

    if (stat (file, &sb) < 0)
        return (0);
    return (S_ISDIR (sb.st_mode));
130 }

/*
 * Returns non-zero if the argument file is a symbolic link.
 */
int
islink (file)
    const char *file;
{
#ifdef S_ISLNK
140 struct stat sb;

    if (CVS_LSTAT (file, &sb) < 0)
        return (0);
    return (S_ISLNK (sb.st_mode));
#else
    return (0);
#endif
}

150 /*
 * Returns non-zero if the argument file is a block or
 * character special device.
 */
int
isdevice (file)
    const char *file;
{
    struct stat sb;

160 if (CVS_LSTAT (file, &sb) < 0)
        return (0);
#ifdef S_ISBLK
    if (S_ISBLK (sb.st_mode))
        return 1;
#endif
#ifdef S_ISCHR
    if (S_ISCHR (sb.st_mode))
        return 1;
#endif
170 return 0;
}

/*
 * Returns non-zero if the argument file exists.
 */
int
isfile (file)
    const char *file;
```

```
{
    return isaccessible(file, F_OK);
}

/*
 * Returns non-zero if the argument file is readable.
 */
int
isreadable (file)
    const char *file;
{
    return isaccessible(file, R_OK);
}

/*
 * Returns non-zero if the argument file is writable.
 */
int
iswritable (file)
    const char *file;
{
    return isaccessible(file, W_OK);
}

/*
 * Returns non-zero if the argument file is accessable according to
 * mode.  If compiled with SETXID_SUPPORT also works if cvs has setxid
 * bits set.
 */
int
isaccessible (file, mode)
    const char *file;
    const int mode;
{
#ifdef SETXID_SUPPORT
    struct stat sb;
    int umask = 0;
    int gmask = 0;
    int omask = 0;
    int uid;

    if (stat(file, &sb) == -1)
        return 0;
    if (mode == F_OK)
        return 1;

    uid = geteuid();
    if (uid == 0)           /* superuser */
    {
        if (mode & X_OK)
            return sb.st_mode & (S_IXUSR|S_IXGRP|S_IXOTH);
        else
            return 1;
    }

    if (mode & R_OK)
    {
        umask |= S_IRUSR;
        gmask |= S_IRGRP;
        omask |= S_IROTH;
    }
    if (mode & W_OK)
    {
        umask |= S_IWUSR;
        gmask |= S_IWGRP;
        omask |= S_IWOTH;
    }
    if (mode & X_OK)
    {
        umask |= S_IXUSR;
        gmask |= S_IXGRP;
        omask |= S_IXOTH;
    }

    if (sb.st_uid == uid)
        return (sb.st_mode & umask) == umask;
    else if (sb.st_gid == getegid())
        return (sb.st_mode & gmask) == gmask;
    else
        return (sb.st_mode & omask) == omask;
#else
    return access(file, mode) == 0;
#endif
}

/*
 * Open a file and die if it fails
 */
FILE *
open_file (name, mode)
```

```
    const char *name;
    const char *mode;
{
    FILE *fp;

    if ((fp = fopen (name, mode)) == NULL)
        error (1, errno, "cannot open %s", name);
    return (fp);
}

/*
 * Make a directory and die if it fails
 */
void
make_directory (name)
    const char *name;
{
    struct stat sb;

    if (stat (name, &sb) == 0 && (!S_ISDIR (sb.st_mode)))
        error (0, 0, "%s already exists but is not a directory", name);
    if (!noexec && mkdir (name, 0777) < 0)
        error (1, errno, "cannot make directory %s", name);
}

/*
 * Make a path to the argument directory, printing a message if something
 * goes wrong.
 */
void
make_directories (name)
    const char *name;
{
    char *cp;

    if (noexec)
        return;

    if (mkdir (name, 0777) == 0 || errno == EEXIST)
        return;
    if (! existence_error (errno))
    {
        error (0, errno, "cannot make path to %s", name);
        return;
    }
    if ((cp = strrchr (name, '/')) == NULL)
        return;
    *cp = '\0';
    make_directories (name);
    *cp++ = '/';
    if (*cp == '\0')
        return;
    (void) mkdir (name, 0777);
}

/* Create directory NAME if it does not already exist; fatal error for
   other errors.  Returns 0 if directory was created; 1 if it already
   existed.  */
int
mkdir_if_needed (name)
    char *name;
{
    if (mkdir (name, 0777) < 0)
    {
        if (!(errno == EEXIST
              || (errno == EACCES && isdir (name))))
            error (1, errno, "cannot make directory %s", name);
        return 1;
    }
    return 0;
}

/*
 * Change the mode of a file, either adding write permissions, or removing
 * all write permissions.  Either change honors the current umask setting.
 *
 * Don't do anything if PreservePermissions is set to 'yes'.  This may
 * have unexpected consequences for some uses of xchmod.
 */
void
xchmod (fname, writable)
    char *fname;
    int writable;
{
    struct stat sb;
    mode_t mode, oumask;

    if (preserve_perms)
        return;
```

```
    if (stat (fname, &sb) < 0)
360 {
        if (!noexec)
            error (0, errno, "cannot stat %s", fname);
        return;
    }
    oumask = umask (0);
    (void) umask (oumask);
    if (writable)
    {
        mode = sb.st_mode | (~oumask
370                          & (((sb.st_mode & S_IRUSR) ? S_IWUSR : 0)
                             | ((sb.st_mode & S_IRGRP) ? S_IWGRP : 0)
                             | ((sb.st_mode & S_IROTH) ? S_IWOTH : 0)));
    }
    else
    {
        mode = sb.st_mode & ~(S_IWRITE | S_IWGRP | S_IWOTH) & ~oumask;
    }

    if (trace)
380 #ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> chmod(%s,%o)\n",
                        (server_active) ? 'S' : ' ', fname,
                        (unsigned int) mode);
#else
        (void) fprintf (stderr, "-> chmod(%s,%o)\n", fname,
                        (unsigned int) mode);
#endif
    if (noexec)
        return;
390

    if (chmod (fname, mode) < 0)
        error (0, errno, "cannot change mode of file %s", fname);
}

/*
 * Rename a file and die if it fails
 */
void
rename_file (from, to)
400 const char *from;
    const char *to;
{
    if (trace)
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> rename(%s,%s)\n",
                        (server_active) ? 'S' : ' ', from, to);
#else
        (void) fprintf (stderr, "-> rename(%s,%s)\n", from, to);
#endif
410 if (noexec)
        return;

    if (rename (from, to) < 0)
        error (1, errno, "cannot rename file %s to %s", from, to);
}

/*
 * unlink a file, if possible.
 */
420 int
unlink_file (f)
    const char *f;
{
    if (trace)
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> unlink(%s)\n",
                        (server_active) ? 'S' : ' ', f);
#else
        (void) fprintf (stderr, "-> unlink(%s)\n", f);
430 #endif
    if (noexec)
        return (0);

    return (unlink (f));
}

/*
 * Unlink a file or dir, if possible.  If it is a directory do a deep
 * removal of all of the files in the directory.  Return -1 on error
440 * (in which case errno is set).
 */
int
unlink_file_dir (f)
    const char *f;
{
    struct stat sb;

    if (trace
```

```
#ifdef SERVER_SUPPORT
        /* This is called by the server parent process in contexts where
           it is not OK to send output (e.g. after we sent "ok" to the
           client).  */
        && !server_active
#endif
        )
        (void) fprintf (stderr, "-> unlink_file_dir(%s)\n", f);

    if (noexec)
        return (0);

    /* For at least some unices, if root tries to unlink() a directory,
       instead of doing something rational like returning EISDIR,
       the system will gleefully go ahead and corrupt the filesystem.
       So we first call stat() to see if it is OK to call unlink().  This
       doesn't quite work--if someone creates a directory between the
       call to stat() and the call to unlink(), we'll still corrupt
       the filesystem.  Where is the Unix Haters Handbook when you need
       it?  */
    if (stat (f, &sb) < 0)
    {
        if (existence_error (errno))
        {
            /* The file or directory doesn't exist anyhow.  */
            return -1;
        }
    }
    else if (S_ISDIR (sb.st_mode))
        return deep_remove_dir (f);

    return unlink (f);
}

/* Remove a directory and everything it contains.  Returns 0 for
 * success, -1 for failure (in which case errno is set).
 */

static int
deep_remove_dir (path)
    const char *path;
{
    DIR          *dirp;
    struct dirent *dp;

    if (rmdir (path) != 0)
    {
        if (errno == ENOTEMPTY
            || errno == EEXIST
            /* Ugly workaround for ugly AIX 4.1 (and 3.2) header bug
               (it defines ENOTEMPTY and EEXIST to 17 but actually
               returns 87).  */
            || (ENOTEMPTY == 17 && EEXIST == 17 && errno == 87))
        {
            if ((dirp = opendir (path)) == NULL)
                /* If unable to open the directory return
                 * an error
                 */
                return -1;

            while ((dp = readdir (dirp)) != NULL)
            {
                char *buf;

                if (strcmp (dp->d_name, ".") == 0 ||
                        strcmp (dp->d_name, "..") == 0)
                    continue;

                buf = xmalloc (strlen (path) + strlen (dp->d_name) + 5);
                sprintf (buf, "%s/%s", path, dp->d_name);

                /* See comment in unlink_file_dir explanation of why we use
                   isdir instead of just calling unlink and checking the
                   status.  */
                if (isdir(buf))
                {
                    if (deep_remove_dir(buf))
                    {
                        closedir(dirp);
                        free (buf);
                        return -1;
                    }
                }
                else
                {
                    if (unlink (buf) != 0)
                    {
                        closedir(dirp);
                        free (buf);
                        return -1;
```

```
                }
            }
            free (buf);
        }
        closedir (dirp);
        return rmdir (path);
    }
    else
        return -1;
}

/* Was able to remove the directory return 0 */
return 0;
}

/* Read NCHARS bytes from descriptor FD into BUF.
   Return the number of characters successfully read.
   The number returned is always NCHARS unless end-of-file or error.  */
static size_t
block_read (fd, buf, nchars)
    int fd;
    char *buf;
    size_t nchars;
{
    char *bp = buf;
    size_t nread;

    do
    {
        nread = read (fd, bp, nchars);
        if (nread == (size_t)-1)
        {
#ifdef EINTR
            if (errno == EINTR)
                continue;
#endif
            return (size_t)-1;
        }

        if (nread == 0)
            break;

        bp += nread;
        nchars -= nread;
    } while (nchars != 0);

    return bp - buf;
}


/*
 * Compare "file1" to "file2". Return non-zero if they don't compare exactly.
 * If FILE1 and FILE2 are special files, compare their salient characteristics
 * (i.e. major/minor device numbers, links, etc.
 */
int
xcmp (file1, file2)
    const char *file1;
    const char *file2;
{
    char *buf1, *buf2;
    struct stat sb1, sb2;
    int fd1, fd2;
    int ret;

    if (CVS_LSTAT (file1, &sb1) < 0)
        error (1, errno, "cannot lstat %s", file1);
    if (CVS_LSTAT (file2, &sb2) < 0)
        error (1, errno, "cannot lstat %s", file2);

    /* If FILE1 and FILE2 are not the same file type, they are unequal. */
    if ((sb1.st_mode & S_IFMT) != (sb2.st_mode & S_IFMT))
        return 1;

    /* If FILE1 and FILE2 are symlinks, they are equal if they point to
       the same thing. */
    if (S_ISLNK (sb1.st_mode) && S_ISLNK (sb2.st_mode))
    {
        int result;
        buf1 = xreadlink (file1);
        buf2 = xreadlink (file2);
        result = (strcmp (buf1, buf2) == 0);
        free (buf1);
        free (buf2);
        return result;
    }

    /* If FILE1 and FILE2 are devices, they are equal if their device
       numbers match. */
    if (S_ISBLK (sb1.st_mode) || S_ISCHR (sb1.st_mode))
```

```
    {
630     if (sb1.st_rdev == sb2.st_rdev)
            return 0;
        else
            return 1;
    }

    if ((fd1 = open (file1, O_RDONLY)) < 0)
        error (1, errno, "cannot open file %s for comparing", file1);
    if ((fd2 = open (file2, O_RDONLY)) < 0)
        error (1, errno, "cannot open file %s for comparing", file2);
640
    /* A generic file compare routine might compare st_dev & st_ino here
       to see if the two files being compared are actually the same file.
       But that won't happen in CVS, so we won't bother. */

    if (sb1.st_size != sb2.st_size)
        ret = 1;
    else if (sb1.st_size == 0)
        ret = 0;
    else
650 {
        /* FIXME: compute the optimal buffer size by computing the least
           common multiple of the files st_blocks field */
        size_t buf_size = 8 * 1024;
        size_t read1;
        size_t read2;

        buf1 = xmalloc (buf_size);
        buf2 = xmalloc (buf_size);

660     do
        {
            read1 = block_read (fd1, buf1, buf_size);
            if (read1 == (size_t)-1)
                error (1, errno, "cannot read file %s for comparing", file1);

            read2 = block_read (fd2, buf2, buf_size);
            if (read2 == (size_t)-1)
                error (1, errno, "cannot read file %s for comparing", file2);

670         /* assert (read1 == read2); */

            ret = memcmp(buf1, buf2, read1);
        } while (ret == 0 && read1 == buf_size);

        free (buf1);
        free (buf2);
    }

    (void) close (fd1);
680 (void) close (fd2);
    return (ret);
}

/* Generate a unique temporary filename.  Returns a pointer to a newly
   malloc'd string containing the name.  Returns successfully or not at
   all.  */
/* There are at least three functions for generating temporary
   filenames.  We use tempnam (SVID 3) if possible, else mktemp (BSD
   4.3), and as last resort tmpnam (POSIX).  Reason is that tempnam and
690 mktemp both allow to specify the directory in which the temporary
   file will be created.  */
#ifdef HAVE_TEMPNAM
char *
cvs_temp_name ()
{
    char *retval;

    retval = tempnam (Tmpdir, "cvs");
    if (retval == NULL)
700     error (1, errno, "cannot generate temporary filename");
    /* tempnam returns a pointer to a newly malloc'd string, so there's
       no need for a xstrdup  */
    return retval;
}
#else
char *
cvs_temp_name ()
{
#  ifdef HAVE_MKTEMP
710 char *value;
    char *retval;

    value = xmalloc (strlen (Tmpdir) + 40);
    sprintf (value, "%s/%s", Tmpdir, "cvsXXXXXX" );
    retval = mktemp (value);

    if (retval == NULL)
        error (1, errno, "cannot generate temporary filename");
```

```
    return value;
# else
    char value[L_tmpnam + 1];
    char *retval;

    retval = tmpnam (value);
    if (retval == NULL)
        error (1, errno, "cannot generate temporary filename");
    return xstrdup (value);
# endif
}
#endif

/* Return non-zero iff FILENAME is absolute.
   Trivial under Unix, but more complicated under other systems.  */
int
isabsolute (filename)
    const char *filename;
{
    return filename[0] == '/';
}

/*
 * Return a string (dynamically allocated) with the name of the file to which
 * LINK is symlinked.
 */
char *
xreadlink (link)
    const char *link;
{
    char *file = NULL;
    char *tfile;
    int buflen = 128;
    int link_name_len;

    if (!islink (link))
        return NULL;

    /* Get the name of the file to which `from' is linked.
       FIXME: what portability issues arise here?  Are readlink &
       ENAMETOOLONG defined on all systems? -twp */
    do
    {
        file = xrealloc (file, buflen);
        link_name_len = readlink (link, file, buflen - 1);
        buflen *= 2;
    }
    while (link_name_len < 0 && errno == ENAMETOOLONG);

    if (link_name_len < 0)
        error (1, errno, "cannot readlink %s", link);

    file[link_name_len] = '\0';

    tfile = xstrdup (file);
    free (file);

    return tfile;
}


/* Return a pointer into PATH's last component.  */
char *
last_component (path)
    char *path;
{
    char *last = strrchr (path, '/');

    if (last && (last != path))
        return last + 1;
    else
        return path;
}

/* Return the home directory.  Returns a pointer to storage
   managed by this function or its callees (currently getenv).
   This function will return the same thing every time it is
   called.  Returns NULL if there is no home directory.

   Note that for a pserver server, this may return root's home
   directory.  What typically happens is that upon being started from
   inetd, before switching users, the code in cvsrc.c calls
   get_homedir which remembers root's home directory in the static
   variable.  Then the switch happens and get_homedir might return a
   directory that we don't even have read or execute permissions for
   (which is bad, when various parts of CVS try to read there).  One
   fix would be to make the value returned by get_homedir only good
   until the next call (which would free the old value).  Another fix
   would be to just always malloc our answer, and let the caller free
   it (that is best, because some day we may need to be reentrant).
```

```
    The workaround is to put -f in inetd.conf which means that
    get_homedir won't get called until after the switch in user ID.

    The whole concept of a "home directory" on the server is pretty
    iffy, although I suppose some people probably are relying on it for
    .cvsrc and such, in the cases where it works.  */
char *
get_homedir ()
{
    static char *home = NULL;
    char *env = getenv ("HOME");
    struct passwd *pw;

    if (home != NULL)
        return home;

    if (env)
        home = env;
    else if ((pw = (struct passwd *) getpwuid (getuid ()))
             && pw->pw_dir)
        home = xstrdup (pw->pw_dir);
    else
        return 0;

    return home;
}

/* See cvs.h for description.  On unix this does nothing, because the
   shell expands the wildcards.  */
void
expand_wild (argc, argv, pargc, pargv)
    int argc;
    char **argv;
    int *pargc;
    char ***pargv;
{
    int i;
    *pargc = argc;
    *pargv = (char **) xmalloc (argc * sizeof (char *));
    for (i = 0; i < argc; ++i)
        (*pargv)[i] = xstrdup (argv[i]);
}

#ifdef SERVER_SUPPORT
/* Case-insensitive string compare.  I know that some systems
   have such a routine, but I'm not sure I see any reasons for
   dealing with the hair of figuring out whether they do (I haven't
   looked into whether this is a performance bottleneck; I would guess
   not).  */
int
cvs_casecmp (str1, str2)
    char *str1;
    char *str2;
{
    char *p;
    char *q;
    int pqdiff;

    p = str1;
    q = str2;
    while ((pqdiff = tolower (*p) - tolower (*q)) == 0)
    {
        if (*p == '\0')
            return 0;
        ++p;
        ++q;
    }
    return pqdiff;
}

/* Case-insensitive file open.  As you can see, this is an expensive
   call.  We don't regard it as our main strategy for dealing with
   case-insensitivity.  Returns errno code or 0 for success.  Puts the
   new file in *FP.  NAME and MODE are as for fopen.  If PATHP is not
   NULL, then put a malloc'd string containing the pathname as found
   into *PATHP.  *PATHP is only set if the return value is 0.

   Might be cleaner to separate the file finding (which just gives
   *PATHP) from the file opening (which the caller can do).  For one
   thing, might make it easier to know whether to put NAME or *PATHP
   into error messages.  */
int
fopen_case (name, mode, fp, pathp)
    char *name;
    char *mode;
    FILE **fp;
    char **pathp;
{
    struct dirent *dp;
```

```
    DIR *dirp;
    char *dir;
    char *fname;
    char *found_name;
    int retval;

    /* Separate NAME into directory DIR and filename within the directory
       FNAME.  */
    dir = xstrdup (name);
    fname = strrchr (dir, '/');
    if (fname == NULL)
        error (1, 0, "internal error: relative pathname in fopen_case");
    *fname++ = '\0';

    found_name = NULL;
    dirp = CVS_OPENDIR (dir);
    if (dirp == NULL)
    {
        if (existence_error (errno))
        {
            /* This can happen if we are looking in the Attic and the Attic
               directory does not exist.  Return the error to the caller;
               they know what to do with it.  */
            retval = errno;
            goto out;
        }
        else
        {
            /* Give a fatal error; that way the error message can be
               more specific than if we returned the error to the caller.  */
            error (1, errno, "cannot read directory %s", dir);
        }
    }
    errno = 0;
    while ((dp = readdir (dirp)) != NULL)
    {
        if (cvs_casecmp (dp->d_name, fname) == 0)
        {
            if (found_name != NULL)
                error (1, 0, "%s is ambiguous; could mean %s or %s",
                       fname, dp->d_name, found_name);
            found_name = xstrdup (dp->d_name);
        }
    }
    if (errno != 0)
        error (1, errno, "cannot read directory %s", dir);
    closedir (dirp);

    if (found_name == NULL)
    {
        *fp = NULL;
        retval = ENOENT;
    }
    else
    {
        char *p;

        /* Copy the found name back into DIR.  We are assuming that
           found_name is the same length as fname, which is true as
           long as the above code is just ignoring case and not other
           aspects of filename syntax.  */
        p = dir + strlen (dir);
        *p++ = '/';
        strcpy (p, found_name);
        *fp = fopen (dir, mode);
        if (*fp == NULL)
            retval = errno;
        else
            retval = 0;
    }

    if (pathp == NULL)
        free (dir);
    else if (retval != 0)
        free (dir);
    else
        *pathp = dir;
    free (found_name);
 out:
    return retval;
}
#endif /* SERVER_SUPPORT */
```

A.24 find_names.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Find Names
 *
 * Finds all the pertinent file names, both from the administration and from the
 * repository
 *
 * Find Dirs
 *
 * Finds all pertinent sub-directories of the checked out instantiation and the
 * repository (and optionally the attic)
 */

#include "cvs.h"

static int find_dirs PROTO((char *dir, List * list, int checkadm,
                            List *entries));
static int find_rcs PROTO((char *dir, List * list));
static int add_subdir_proc PROTO((Node *, void *));
static int register_subdir_proc PROTO((Node *, void *));

static List *filelist;

/*
 * add the key from entry on entries list to the files list
 */
static int add_entries_proc PROTO((Node *, void *));
static int
add_entries_proc (node, closure)
     Node *node;
     void *closure;
{
    Entnode *entnode;
    Node *fnode;

    entnode = (Entnode *) node->data;
    if (entnode->type != ENT_FILE)
        return (0);

    fnode = getnode ();
    fnode->type = FILES;
    fnode->key = xstrdup (node->key);
    if (addnode (filelist, fnode) != 0)
        freenode (fnode);
    return (0);
}

List *
Find_Names (repository, which, aflag, optentries)
    char *repository;
    int which;
    int aflag;
    List **optentries;
{
    List *entries;
    List *files;

    /* make a list for the files */
    files = filelist = getlist ();

    /* look at entries (if necessary) */
    if (which & W_LOCAL)
    {
        /* parse the entries file (if it exists) */
        entries = Entries_Open (aflag, NULL);
        if (entries != NULL)
        {
            /* walk the entries file adding elements to the files list */
            (void) walklist (entries, add_entries_proc, NULL);

            /* if our caller wanted the entries list, return it; else free it */
            if (optentries != NULL)
                *optentries = entries;
            else
                Entries_Close (entries);
        }
    }

    if ((which & W_REPOS) && repository && !isreadable (CVSADM_ENTSTAT))
    {
        /* search the repository */
        if (find_rcs (repository, files) != 0)
            error (1, errno, "cannot open directory %s", repository);
```

```
90          /* search the attic too */
            if (which & W_ATTIC)
            {
                char *dir;
                dir = xmalloc (strlen (repository) + sizeof (CVSATTIC) + 10);
                (void) sprintf (dir, "%s/%s", repository, CVSATTIC);
                (void) find_rcs (dir, files);
                free (dir);
            }
        }
100
        /* sort the list into alphabetical order and return it */
        sortlist (files, fsortcmp);
        return (files);
    }

    /*
     * Add an entry from the subdirs list to the directories list.  This
     * is called via walklist.
     */
110
    static int
    add_subdir_proc (p, closure)
         Node *p;
         void *closure;
    {
        List *dirlist = (List *) closure;
        Entnode *entnode;
        Node *dnode;

120     entnode = (Entnode *) p->data;
        if (entnode->type != ENT_SUBDIR)
            return 0;

        dnode = getnode ();
        dnode->type = DIRS;
        dnode->key = xstrdup (entnode->user);
        if (addnode (dirlist, dnode) != 0)
            freenode (dnode);
        return 0;
130 }

    /*
     * Register a subdirectory.  This is called via walklist.
     */

    /*ARGSUSED*/
    static int
    register_subdir_proc (p, closure)
         Node *p;
140      void *closure;
    {
        List *entries = (List *) closure;

        Subdir_Register (entries, (char *) NULL, p->key);
        return 0;
    }

    /*
     * create a list of directories to traverse from the current directory
150  */
    List *
    Find_Directories (repository, which, entries)
        char *repository;
        int which;
        List *entries;
    {
        List *dirlist;

        /* make a list for the directories */
160     dirlist = getlist ();

        /* find the local ones */
        if (which & W_LOCAL)
        {
            List *tmpentries;
            struct stickydirtag *sdtp;

            /* Look through the Entries file.  */

170         if (entries != NULL)
                tmpentries = entries;
            else if (isfile (CVSADM_ENT))
                tmpentries = Entries_Open (0, NULL);
            else
                tmpentries = NULL;

            if (tmpentries != NULL)
                sdtp = (struct stickydirtag *) tmpentries->list->data;
```

```
        /* If we do have an entries list, then if sdtp is NULL, or if
           sdtp->subdirs is nonzero, all subdirectory information is
           recorded in the entries list.  */
        if (tmpentries != NULL && (sdtp == NULL || sdtp->subdirs))
            walklist (tmpentries, add_subdir_proc, (void *) dirlist);
        else
        {
            /* This is an old working directory, in which subdirectory
               information is not recorded in the Entries file.  Find
               the subdirectories the hard way, and, if possible, add
               it to the Entries file for next time.  */

            /* FIXME-maybe: find_dirs is bogus for this usage because
               it skips CVSATTIC and CVSLCK directories--those names
               should be special only in the repository.  However, in
               the interests of not perturbing this code, we probably
               should leave well enough alone unless we want to write
               a sanity.sh test case (which would operate by manually
               hacking on the CVS/Entries file).  */

            if (find_dirs (".", dirlist, 1, tmpentries) != 0)
                error (1, errno, "cannot open current directory");
            if (tmpentries != NULL)
            {
                if (! list_isempty (dirlist))
                    walklist (dirlist, register_subdir_proc,
                              (void *) tmpentries);
                else
                    Subdirs_Known (tmpentries);
            }
        }

        if (entries == NULL && tmpentries != NULL)
            Entries_Close (tmpentries);
    }

    /* look for sub-dirs in the repository */
    if ((which & W_REPOS) && repository)
    {
        /* search the repository */
        if (find_dirs (repository, dirlist, 0, entries) != 0)
            error (1, errno, "cannot open directory %s", repository);

        /* We don't need to look in the attic because directories
           never go in the attic.  In the future, there hopefully will
           be a better mechanism for detecting whether a directory in
           the repository is alive or dead; it may or may not involve
           moving directories to the attic.  */
    }

    /* sort the list into alphabetical order and return it */
    sortlist (dirlist, fsortcmp);
    return (dirlist);
}

/*
 * Finds all the ,v files in the argument directory, and adds them to the
 * files list.  Returns 0 for success and non-zero if the argument directory
 * cannot be opened.
 */
static int
find_rcs (dir, list)
    char *dir;
    List *list;
{
    Node *p;
    struct dirent *dp;
    DIR *dirp;

    /* set up to read the dir */
    if ((dirp = CVS_OPENDIR (dir)) == NULL)
        return (1);

    /* read the dir, grabbing the ,v files */
    while ((dp = readdir (dirp)) != NULL)
    {
        if (CVS_FNMATCH (RCSPAT, dp->d_name, 0) == 0)
        {
            char *comma;

            comma = strrchr (dp->d_name, ','); /* strip the ,v */
            *comma = '\0';
            p = getnode ();
            p->type = FILES;
            p->key = xstrdup (dp->d_name);
            if (addnode (list, p) != 0)
                freenode (p);
        }
    }
```

```
    (void) closedir (dirp);
    return (0);
}

/*
 * Finds all the subdirectories of the argument dir and adds them to
 * the specified list.  Sub-directories without a CVS administration
 * directory are optionally ignored.  If ENTRIES is not NULL, all
 * files on the list are ignored.  Returns 0 for success or 1 on
 * error.
 */
static int
find_dirs (dir, list, checkadm, entries)
    char *dir;
    List *list;
    int checkadm;
    List *entries;
{
    Node *p;
    char *tmp = NULL;
    size_t tmp_size = 0;
    struct dirent *dp;
    DIR *dirp;
    int skip_emptydir = 0;

    /* First figure out whether we need to skip directories named
       Emptydir.  Except in the CVSNULLREPOS case, Emptydir is just
       a normal directory name.  */
    if (isabsolute (dir)
        && strncmp (dir, CVSroot_directory, strlen (CVSroot_directory)) == 0
        && ISDIRSEP (dir[strlen (CVSroot_directory)])
        && strcmp (dir + strlen (CVSroot_directory) + 1, CVSROOTADM) == 0)
        skip_emptydir = 1;

    /* set up to read the dir */
    if ((dirp = CVS_OPENDIR (dir)) == NULL)
        return (1);

    /* read the dir, grabbing sub-dirs */
    while ((dp = readdir (dirp)) != NULL)
    {
        if (strcmp (dp->d_name, ".") == 0 ||
            strcmp (dp->d_name, "..") == 0 ||
            strcmp (dp->d_name, CVSATTIC) == 0 ||
            strcmp (dp->d_name, CVSLCK) == 0 ||
            strcmp (dp->d_name, CVSREP) == 0)
            continue;

        /* findnode() is going to be significantly faster than stat()
           because it involves no system calls.  That is why we bother
           with the entries argument, and why we check this first.  */
        if (entries != NULL && findnode (entries, dp->d_name) != NULL)
            continue;

        if (skip_emptydir
            && strcmp (dp->d_name, CVSNULLREPOS) == 0)
            continue;

#ifdef DT_DIR
        if (dp->d_type != DT_DIR)
        {
            if (dp->d_type != DT_UNKNOWN && dp->d_type != DT_LNK)
                continue;
#endif
            /* don't bother stating ,v files */
            if (CVS_FNMATCH (RCSPAT, dp->d_name, 0) == 0)
                continue;

            expand_string (&tmp,
                           &tmp_size,
                           strlen (dir) + strlen (dp->d_name) + 10);
            sprintf (tmp, "%s/%s", dir, dp->d_name);
            if (!isdir (tmp))
                continue;

#ifdef DT_DIR
        }
#endif

        /* check for administration directories (if needed) */
        if (checkadm)
        {
            /* blow off symbolic links to dirs in local dir */
#ifdef DT_DIR
            if (dp->d_type != DT_DIR)
            {
                /* we're either unknown or a symlink at this point */
                if (dp->d_type == DT_LNK)
                    continue;
#endif
```

```
                /* Note that we only get here if we already set tmp
                   above.  */
                if (islink (tmp))
                    continue;
#ifdef DT_DIR
            }
#endif

            /* check for new style */
            expand_string (&tmp,
                           &tmp_size,
                           (strlen (dir) + strlen (dp->d_name)
                            + sizeof (CVSADM) + 10));
            (void) sprintf (tmp, "%s/%s/%s", dir, dp->d_name, CVSADM);
            if (!isdir (tmp))
                continue;
        }

        /* put it in the list */
        p = getnode ();
        p->type = DIRS;
        p->key = xstrdup (dp->d_name);
        if (addnode (list, p) != 0)
            freenode (p);
    }
    (void) closedir (dirp);
    if (tmp != NULL)
        free (tmp);
    return (0);
}
```

A.25 hardlink.c

```
/* This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

/* Collect and manage hardlink info associated with a particular file.  */

#include "cvs.h"
#include "hardlink.h"

/* The structure currently used to manage hardlink info is a list.
   Therefore, most of the functions which manipulate hardlink data
   are walklist procedures.  This is not a very efficient implementation;
   if someone decides to use a real hash table (for instance), then
   much of this code can be rewritten to be a little less arcane.

   Each element of 'hardlist' represents an inode.  It is keyed on the
   inode number, and points to a list of files.  This is to make it
   easy to find out what files are linked to a given file FOO: find
   FOO's inode, look it up in hardlist, and retrieve the list of files
   associated with that inode.

   Each file node, in turn, is represented by a 'hardlink_info' struct,
   which includes 'status' and 'links' fields.  The 'status' field should
   be used by a procedure like commit_fileproc or update_fileproc to
   record each file's status; that way, after all file links have been
   recorded, CVS can check the linkage of files which are in doubt
   (i.e. T_NEEDS_MERGE files).

   TODO: a diagram of an example hardlist would help here. */

/* TODO: change this to something with a marginal degree of
   efficiency, like maybe a hash table.  Yeah. */

List *hardlist;      /* Record hardlink information for working files */
char *working_dir; /* The top-level working directory, used for
                      constructing full pathnames. */

/* Return a pointer to FILEPATH's node in the hardlist.  This means
   looking up its inode, retrieving the list of files linked to that
   inode, and then looking up FILE in that list.  If the file doesn't
   seem to exist, return NULL. */
Node *
lookup_file_by_inode (filepath)
    const char *filepath;
{
    char *inodestr, *file;
    struct stat sb;
    Node *hp, *p;

    /* Get file's basename, so that we can stat it. */
    file = strrchr (filepath, '/');
    if (file)
        ++file;
    else
        file = (char *) filepath;

    /* inodestr contains the hexadecimal representation of an
       inode, so it requires two bytes of text to represent
       each byte of the inode number. */
    inodestr = (char *) xmalloc (2*sizeof(ino_t)*sizeof(char) + 1);
    if (stat (file, &sb) < 0)
    {
        if (errno == ENOENT)
        {
            /* The file doesn't exist; we may be doing an update on a
               file that's been removed.  A nonexistent file has no
               link information, so return without changing hardlist. */
            free (inodestr);
            return NULL;
        }
        error (1, errno, "cannot stat %s", file);
    }

    sprintf (inodestr, "%lx", (unsigned long) sb.st_ino);

    /* Find out if this inode is already in the hardlist, adding
       a new entry to the list if not. */
    hp = findnode (hardlist, inodestr);
    if (hp == NULL)
    {
        hp = getnode ();
        hp->type = UNKNOWN;
```

```
        hp->key = inodestr;
        hp->data = (char *) getlist();
        hp->delproc = dellist;
        (void) addnode (hardlist, hp);
    }
    else
    {
        free (inodestr);
    }

    p = findnode ((List *) hp->data, filepath);
    if (p == NULL)
    {
        p = getnode();
        p->type = UNKNOWN;
        p->key = xstrdup (filepath);
        p->data = NULL;
        (void) addnode ((List *) hp->data, p);
    }

    return p;
}

/* After a file has been checked out, add a node for it to the hardlist
   (if necessary) and mark it as checked out. */
void
update_hardlink_info (file)
    const char *file;
{
    char *path;
    Node *n;
    struct hardlink_info *hlinfo;

    if (file[0] == '/')
    {
        path = xstrdup (file);
    }
    else
    {
        /* file is a relative pathname; assume it's from the current
           working directory. */
        char *dir = xgetwd();
        path = xmalloc (sizeof(char) * (strlen(dir) + strlen(file) + 2));
        sprintf (path, "%s/%s", dir, file);
        free (dir);
    }

    n = lookup_file_by_inode (path);
    if (n == NULL)
    {
        /* Something is *really* wrong if the file doesn't exist here;
           update_hardlink_info should be called only when a file has
           just been checked out to a working directory. */
        error (1, 0, "lost hardlink info for %s", file);
    }

    if (n->data == NULL)
        n->data = (char *) xmalloc (sizeof (struct hardlink_info));
    hlinfo = (struct hardlink_info *) n->data;
    hlinfo->status = T_UPTODATE;
    hlinfo->checked_out = 1;
}

/* Return a List with all the files known to be linked to FILE in
   the working directory.  Used by special_file_mismatch, to determine
   whether it is safe to merge two files.

   FIXME: What is the memory allocation for the return value?  We seem
   to sometimes allocate a new list (getlist() call below) and sometimes
   return an existing list (where we return n->data).  */
List *
list_linked_files_on_disk (file)
    char *file;
{
    char *inodestr, *path;
    struct stat sb;
    Node *n;

    /* If hardlist is NULL, we have not been doing an operation that
       would permit us to know anything about the file's hardlinks
       (cvs update, cvs commit, etc).  Return an empty list. */
    if (hardlist == NULL)
        return getlist();

    /* Get the full pathname of file (assuming the working directory) */
    if (file[0] == '/')
        path = xstrdup (file);
    else
    {
        char *dir = xgetwd();
```

```
        path = (char *) xmalloc (sizeof(char) *
                                 (strlen(dir) + strlen(file) + 2));
        sprintf (path, "%s/%s", dir, file);
        free (dir);
    }

    /* We do an extra lookup_file here just to make sure that there
       is a node for 'path' in the hardlist.  If that were not so,
       comparing the working directory linkage against the repository
       linkage for a file would always fail. */
    (void) lookup_file_by_inode (path);

    if (stat (path, &sb) < 0)
        error (1, errno, "cannot stat %s", file);
    /* inodestr contains the hexadecimal representation of an
       inode, so it requires two bytes of text to represent
       each byte of the inode number. */
    inodestr = (char *) xmalloc (2*sizeof(ino_t)*sizeof(char) + 1);
    sprintf (inodestr, "%lx", (unsigned long) sb.st_ino);

    /* Make sure the files linked to this inode are sorted. */
    n = findnode (hardlist, inodestr);
    sortlist ((List *) n->data, fsortcmp);

    free (inodestr);
    return (List *) n->data;
}

/* Compare the files in the 'key' fields of two lists, returning 1 if
   the lists are equivalent and 0 otherwise.

   Only the basenames of each file are compared. This is an awful hack
   that exists because list_linked_files_on_disk returns full paths
   and the 'hardlinks' structure of a RCSVers node contains only
   basenames.  That in turn is a result of the awful hack that only
   basenames are stored in the RCS file.  If anyone ever solves the
   problem of correctly managing cross-directory hardlinks, this
   function (along with most functions in this file) must be fixed. */

int
compare_linkage_lists (links1, links2)
    List *links1;
    List *links2;
{
    Node *n1, *n2;
    char *p1, *p2;

    sortlist (links1, fsortcmp);
    sortlist (links2, fsortcmp);

    n1 = links1->list->next;
    n2 = links2->list->next;

    while (n1 != links1->list && n2 != links2->list)
    {
        /* Get the basenames of both files. */
        p1 = strrchr (n1->key, '/');
        if (p1 == NULL)
            p1 = n1->key;
        else
            ++p1;

        p2 = strrchr (n2->key, '/');
        if (p2 == NULL)
            p2 = n2->key;
        else
            ++p2;

        /* Compare the files' basenames. */
        if (strcmp (p1, p2) != 0)
            return 0;

        n1 = n1->next;
        n2 = n2->next;
    }

    /* At this point we should be at the end of both lists; if not,
       one file has more links than the other, and return 1. */
    return (n1 == links1->list && n2 == links2->list);
}

/* Find a checked-out file in a list of filenames.  Used by RCS_checkout
   when checking out a new hardlinked file, to decide whether this file
   can be linked to any others that already exist.  The return value
   is not currently used. */

int
find_checkedout_proc (node, data)
    Node *node;
    void *data;
```

```
{
    Node **uptodate = (Node **) data;
    Node *link;
    char *dir = xgetwd();
    char *path;
    struct hardlink_info *hlinfo;

    /* If we have already found a file, don't do anything. */
    if (*uptodate != NULL)
        return 0;

    /* Look at this file in the hardlist and see whether the checked_out
       field is 1, meaning that it has been checked out during this CVS run. */
    path = (char *)
        xmalloc (sizeof(char) * (strlen (dir) + strlen (node->key) + 2));
    sprintf (path, "%s/%s", dir, node->key);
    link = lookup_file_by_inode (path);
    free (path);
    free (dir);

    if (link == NULL)
    {
        /* We haven't seen this file - maybe it hasn't been checked
           out yet at all. */
        return 0;
    }

    hlinfo = (struct hardlink_info *) link->data;
    if (hlinfo->checked_out)
    {
        /* This file has been checked out recently, so it's safe to
           link to it. */
        *uptodate = link;
    }

    return 0;
}
```

A.26 hardlink.h

```
/* This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

/* Data type definitions and declarations for hardlink management.  */

/* This file should be #included in CVS source files after cvs.h
   since it relies on types and macros defined there. */

/* The 'checked_out' member of a hardlink_info struct is used only
   when files are being checked out or updated.  It is used only when
   hardlinked files are being checked out. */

struct hardlink_info
{
    Ctype status;           /* as returned from Classify_File() */
    int checked_out;        /* has this file been checked out lately? */
};

extern List *hardlist;
extern char *working_dir;

Node *lookup_file_by_inode PROTO ((const char *));
void update_hardlink_info PROTO ((const char *));
List *list_linked_files_on_disk PROTO ((char *));
int compare_linkage_lists PROTO ((List *, List *));
int find_checkedout_proc PROTO ((Node *, void *));
```

A.27 hash.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Polk's hash list manager.  So cool.
 */

#include "cvs.h"
#include <assert.h>

/* Global caches.  The idea is that we maintain a linked list of "free"d
   nodes or lists, and get new items from there.  It has been suggested
   to use an obstack instead, but off the top of my head, I'm not sure
   that would gain enough to be worth worrying about.  */
static List *listcache = NULL;
static Node *nodecache = NULL;

static void freenode_mem PROTO((Node * p));

/* hash function */
static int
hashp (key)
    const char *key;
{
    unsigned int h = 0;
    unsigned int g;

    assert(key != NULL);

    while (*key != 0)
    {
        unsigned int c = *key++;
        /* The FOLD_FN_CHAR is so that findnode_fn works.  */
        h = (h << 4) + FOLD_FN_CHAR (c);
        if ((g = h & 0xf0000000) != 0)
            h = (h ^ (g >> 24)) ^ g;
    }

    return (h % HASHSIZE);
}

/*
 * create a new list (or get an old one from the cache)
 */
List *
getlist ()
{
    int i;
    List *list;
    Node *node;

    if (listcache != NULL)
    {
        /* get a list from the cache and clear it */
        list = listcache;
        listcache = listcache->next;
        list->next = (List *) NULL;
        for (i = 0; i < HASHSIZE; i++)
            list->hasharray[i] = (Node *) NULL;
    }
    else
    {
        /* make a new list from scratch */
        list = (List *) xmalloc (sizeof (List));
        memset ((char *) list, 0, sizeof (List));
        node = getnode ();
        list->list = node;
        node->type = HEADER;
        node->next = node->prev = node;
    }
    return (list);
}

/*
 * free up a list
 */
void
dellist (listp)
    List **listp;
{
    int i;
    Node *p;

    if (*listp == (List *) NULL)
        return;
```

```
    p = (*listp)->list;

    /* free each node in the list (except header) */
    while (p->next != p)
        delnode (p->next);

    /* free any list-private data, without freeing the actual header */
    freenode_mem (p);

    /* free up the header nodes for hash lists (if any) */
    for (i = 0; i < HASHSIZE; i++)
    {
        if ((p = (*listp)->hasharray[i]) != (Node *) NULL)
        {
            /* put the nodes into the cache */
#ifndef NOCACHE
            p->type = UNKNOWN;
            p->next = nodecache;
            nodecache = p;
#else
            /* If NOCACHE is defined we turn off the cache.  This can make
               it easier to tools to determine where items were allocated
               and freed, for tracking down memory leaks and the like.  */
            free (p);
#endif
        }
    }

    /* put it on the cache */
#ifndef NOCACHE
    (*listp)->next = listcache;
    listcache = *listp;
#else
    free ((*listp)->list);
    free (*listp);
#endif
    *listp = (List *) NULL;
}

/*
 * get a new list node
 */
Node *
getnode ()
{
    Node *p;

    if (nodecache != (Node *) NULL)
    {
        /* get one from the cache */
        p = nodecache;
        nodecache = p->next;
    }
    else
    {
        /* make a new one */
        p = (Node *) xmalloc (sizeof (Node));
    }

    /* always make it clean */
    memset ((char *) p, 0, sizeof (Node));
    p->type = UNKNOWN;

    return (p);
}

/*
 * remove a node from it's list (maybe hash list too) and free it
 */
void
delnode (p)
    Node *p;
{
    if (p == (Node *) NULL)
        return;

    /* take it out of the list */
    p->next->prev = p->prev;
    p->prev->next = p->next;

    /* if it was hashed, remove it from there too */
    if (p->hashnext != (Node *) NULL)
    {
        p->hashnext->hashprev = p->hashprev;
        p->hashprev->hashnext = p->hashnext;
    }

    /* free up the storage */
    freenode (p);
}
```

```
/*
 * free up the storage associated with a node
 */
static void
freenode_mem (p)
    Node *p;
{
    if (p->delproc != (void (*) ()) NULL)
        p->delproc (p);             /* call the specified delproc */
    else
    {
        if (p->data != NULL)    /* otherwise free() it if necessary */
            free (p->data);
    }
    if (p->key != NULL)         /* free the key if necessary */
        free (p->key);

    /* to be safe, re-initialize these */
    p->key = p->data = (char *) NULL;
    p->delproc = (void (*) ()) NULL;
}

/*
 * free up the storage associated with a node and recycle it
 */
void
freenode (p)
    Node *p;
{
    /* first free the memory */
    freenode_mem (p);

    /* then put it in the cache */
#ifndef NOCACHE
    p->type = UNKNOWN;
    p->next = nodecache;
    nodecache = p;
#else
    free (p);
#endif
}

/*
 * Link item P into list LIST before item MARKER.  If P->KEY is non-NULL and
 * that key is already in the hash table, return -1 without modifying any
 * parameter.
 *
 * return 0 on success
 */
int
insert_before (list, marker, p)
    List *list;
    Node *marker;
    Node *p;
{
    if (p->key != NULL)         /* hash it too? */
    {
        int hashval;
        Node *q;

        hashval = hashp (p->key);
        if (list->hasharray[hashval] == NULL) /* make a header for list? */
        {
            q = getnode ();
            q->type = HEADER;
            list->hasharray[hashval] = q->hashnext = q->hashprev = q;
        }

        /* put it into the hash list if it's not already there */
        for (q = list->hasharray[hashval]->hashnext;
             q != list->hasharray[hashval]; q = q->hashnext)
        {
            if (strcmp (p->key, q->key) == 0)
                return (-1);
        }
        q = list->hasharray[hashval];
        p->hashprev = q->hashprev;
        p->hashnext = q;
        p->hashprev->hashnext = p;
        q->hashprev = p;
    }

    p->next = marker;
    p->prev = marker->prev;
    marker->prev->next = p;
    marker->prev = p;

    return (0);
}
```

```
/*
 * insert item p at end of list "list" (maybe hash it too) if hashing and it
 * already exists, return -1 and don't actually put it in the list
 *
 * return 0 on success
 */
int
addnode (list, p)
    List *list;
    Node *p;
{
  return insert_before (list, list->list, p);
}

/*
 * Like addnode, but insert p at the front of 'list'.  This bogosity is
 * necessary to preserve last-to-first output order for some RCS functions.
 */
int
addnode_at_front (list, p)
    List *list;
    Node *p;
{
  return insert_before (list, list->list->next, p);
}

/* Look up an entry in hash list table and return a pointer to the
   node.  Return NULL if not found.  Abort with a fatal error for
   errors.  */
Node *
findnode (list, key)
    List *list;
    const char *key;
{
    Node *head, *p;

    /* This probably should be "assert (list != NULL)" (or if not we
       should document the current behavior), but only if we check all
       the callers to see if any are relying on this behavior.  */
    if ((list == (List *) NULL))
        return ((Node *) NULL);

    assert (key != NULL);

    head = list->hasharray[hashp (key)];
    if (head == (Node *) NULL)
        /* Not found.  */
        return ((Node *) NULL);

    for (p = head->hashnext; p != head; p = p->hashnext)
        if (strcmp (p->key, key) == 0)
            return (p);
    return ((Node *) NULL);
}

/*
 * Like findnode, but for a filename.
 */
Node *
findnode_fn (list, key)
    List *list;
    const char *key;
{
    Node *head, *p;

    /* This probably should be "assert (list != NULL)" (or if not we
       should document the current behavior), but only if we check all
       the callers to see if any are relying on this behavior.  */
    if (list == (List *) NULL)
        return ((Node *) NULL);

    assert (key != NULL);

    head = list->hasharray[hashp (key)];
    if (head == (Node *) NULL)
        return ((Node *) NULL);

    for (p = head->hashnext; p != head; p = p->hashnext)
        if (fncmp (p->key, key) == 0)
            return (p);
    return ((Node *) NULL);
}

/*
 * walk a list with a specific proc
 */
int
walklist (list, proc, closure)
    List *list;
```

```
    int (*proc) PROTO ((Node *, void *));
    void *closure;
{
    Node *head, *p;
    int err = 0;

    if (list == NULL)
        return (0);

    head = list->list;
    for (p = head->next; p != head; p = p->next)
        err += proc (p, closure);
    return (err);
}

int
list_isempty (list)
    List *list;
{
    return list == NULL || list->list->next == list->list;
}

static int (*client_comp) PROTO ((const Node *, const Node *));
static int qsort_comp PROTO ((const void *, const void *));

static int
qsort_comp (elem1, elem2)
    const void *elem1;
    const void *elem2;
{
    Node **node1 = (Node **) elem1;
    Node **node2 = (Node **) elem2;
    return client_comp (*node1, *node2);
}

/*
 * sort the elements of a list (in place)
 */
void
sortlist (list, comp)
    List *list;
    int (*comp) PROTO ((const Node *, const Node *));
{
    Node *head, *remain, *p, **array;
    int i, n;

    /* save the old first element of the list */
    head = list->list;
    remain = head->next;

    /* count the number of nodes in the list */
    n = 0;
    for (p = remain; p != head; p = p->next)
        n++;

    /* allocate an array of nodes and populate it */
    array = (Node **) xmalloc (sizeof(Node *) * n);
    i = 0;
    for (p = remain; p != head; p = p->next)
        array[i++] = p;

    /* sort the array of nodes */
    client_comp = comp;
    qsort (array, n, sizeof(Node *), qsort_comp);

    /* rebuild the list from beginning to end */
    head->next = head->prev = head;
    for (i = 0; i < n; i++)
    {
        p = array[i];
        p->next = head;
        p->prev = head->prev;
        p->prev->next = p;
        head->prev = p;
    }

    /* release the array of nodes */
    free (array);
}

/*
 * compare two files list node (for sort)
 */
int
fsortcmp (p, q)
    const Node *p;
    const Node *q;
{
    return (strcmp (p->key, q->key));
}
```

```
/* Debugging functions.  Quite useful to call from within gdb. */

static char *nodetypestring PROTO ((Ntype));

static char *
nodetypestring (type)
    Ntype type;
{
    switch (type) {
    case UNKNOWN: return("UNKNOWN");
    case HEADER: return("HEADER");
    case ENTRIES: return("ENTRIES");
    case FILES:     return("FILES");
    case LIST:      return("LIST");
    case RCSNODE: return("RCSNODE");
    case RCSVERS: return("RCSVERS");
    case DIRS:      return("DIRS");
    case UPDATE: return("UPDATE");
    case LOCK:      return("LOCK");
    case NDBMNODE: return("NDBMNODE");
    case FILEATTR: return("FILEATTR");
    case VARIABLE: return("VARIABLE");
    case RCSFIELD: return("RCSFIELD");
    }

    return("<trash>");
}

static int printnode PROTO ((Node *, void *));
static int
printnode (node, closure)
     Node *node;
     void *closure;
{
    if (node == NULL)
    {
        (void) printf("NULL node.\n");
        return(0);
    }

    (void) printf("Node at 0x%p: type = %s, key = 0x%p = \"%s\", data = 0x%p, next = 0x%p, prev = 0x%p\n",
            node, nodetypestring(node->type), node->key, node->key, node->data, node->next, node->prev);

    return(0);
}

/* This is global, not static, so that its name is unique and to avoid
   compiler warnings about it not being used.  But it is not used by CVS;
   it exists so one can call it from a debugger.  */
void printlist PROTO ((List *));

void
printlist (list)
    List *list;
{
    if (list == NULL)
    {
        (void) printf("NULL list.\n");
        return;
    }

    (void) printf("List at 0x%p: list = 0x%p, HASHSIZE = %d, next = 0x%p\n",
            list, list->list, HASHSIZE, list->next);

    (void) walklist(list, printnode, NULL);

    return;
}
```

A.28 hash.h

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

/*
 * The number of buckets for the hash table contained in each list.  This
 * should probably be prime.
 */
#define HASHSIZE 151

/*
 * Types of nodes
 */
enum ntype
{
    UNKNOWN, HEADER, ENTRIES, FILES, LIST, RCSNODE,
    RCSVERS, DIRS, UPDATE, LOCK, NDBMNODE, FILEATTR,
    VARIABLE, RCSFIELD
};
typedef enum ntype Ntype;

struct node
{
    Ntype type;
    struct node *next;
    struct node *prev;
    struct node *hashnext;
    struct node *hashprev;
    char *key;
    char *data;
    void (*delproc) ();
};
typedef struct node Node;

struct list
{
    Node *list;
    Node *hasharray[HASHSIZE];
    struct list *next;
};
typedef struct list List;

List *getlist PROTO((void));
Node *findnode PROTO((List * list, const char *key));
Node *findnode_fn PROTO((List * list, const char *key));
Node *getnode PROTO((void));
int insert_before PROTO((List * list, Node * marker, Node * p));
int addnode PROTO((List * list, Node * p));
int addnode_at_front PROTO((List * list, Node * p));
int walklist PROTO((List * list, int (*)(Node *n, void *closure), void *closure));
int list_isempty PROTO ((List *list));
void dellist PROTO((List ** listp));
void delnode PROTO((Node * p));
void freenode PROTO((Node * p));
void sortlist PROTO((List * list, int (*)(const Node *, const Node *)));
int fsortcmp PROTO((const Node * p, const Node * q));
```

A.29 history.c

```
/*
 *
 *    You may distribute under the terms of the GNU General Public License
 *    as specified in the README file that comes with the CVS 1.0 kit.
 *
 * ********** History of Users and Module **********
 *
 * LOGGING:  Append record to "${CVSROOT}/CVSROOTADM/CVSROOTADM_HISTORY".
 *
 * On For each Tag, Add, Checkout, Commit, Update or Release command,
 * one line of text is written to a History log.
 *
 *     X date | user | CurDir | special | rev(s) | argument '\n'
 *
 * where: [The spaces in the example line above are not in the history file.]
 *
 *  X          is a single character showing the type of event:
 *             T       "Tag" cmd.
 *             O       "Checkout" cmd.
 *             E       "Export" cmd.
 *             F       "Release" cmd.
 *             W       "Update" cmd - No User file, Remove from Entries file.
 *             U       "Update" cmd - File was checked out over User file.
 *             G       "Update" cmd - File was merged successfully.
 *             C       "Update" cmd - File was merged and shows overlaps.
 *             M       "Commit" cmd - "Modified" file.
 *             A       "Commit" cmd - "Added" file.
 *             R       "Commit" cmd - "Removed" file.
 *
 *  date       is a fixed length 8-char hex representation of a Unix time_t.
 *             [Starting here, variable fields are delimited by '|' chars.]
 *
 *  user       is the username of the person who typed the command.
 *
 *  CurDir     The directory where the action occurred.  This should be the
 *             absolute path of the directory which is at the same level as
 *             the "Repository" field (for W,U,G,C & M,A,R).
 *
 *  Repository For record types [W,U,G,C,M,A,R] this field holds the
 *             repository read from the administrative data where the
 *             command was typed.
 *             T       "A" -> New Tag, "D" -> Delete Tag
 *                     Otherwise it is the Tag or Date to modify.
 *             O,F,E A "" (null field)
 *
 *  rev(s)     Revision number or tag.
 *             T       The Tag to apply.
 *             O,E     The Tag or Date, if specified, else "" (null field).
 *             F       "" (null field)
 *             W       The Tag or Date, if specified, else "" (null field).
 *             U       The Revision checked out over the User file.
 *             G,C     The Revision(s) involved in merge.
 *             M,A,R RCS Revision affected.
 *
 *  argument The module (for [TOEUF]) or file (for [WUGCMAR]) affected.
 *
 *
 *** Report categories: "User" and "Since" modifiers apply to all reports.
 *                     [For "sort" ordering see the "sort_order" routine.]
 *
 *   Extract list of record types
 *
 *     -e, -x [TOEFWUGCMAR]
 *
 *             Extracted records are simply printed, No analysis is performed.
 *             All "field" modifiers apply.  -e chooses all types.
 *
 *   Checked 'O'ut modules
 *
 *     -o, -w
 *             Checked out modules.  'F' and 'O' records are examined and if
 *             the last record for a repository/file is an 'O', a line is
 *             printed.  "-w" forces the "working dir" to be used in the
 *             comparison instead of the repository.
 *
 *   Committed (Modified) files
 *
 *     -c, -l, -w
 *             All 'M'odified, 'A'dded and 'R'emoved records are examined.
 *             "Field" modifiers apply.  -l forces a sort by file within user
 *             and shows only the last modifier.  -w works as in Checkout.
 *
 *             Warning: Be careful with what you infer from the output of
 *                      "cvs hi -c -l".  It means the last time *you*
 *                      changed the file, not the list of files for which
 *                      you were the last changer!!!
 *
 *   Module history for named modules.
```

```
 *     -m module, -l
 *
 *          This is special.  If one or more modules are specified, the
 *          module names are remembered and the files making up the
 *          modules are remembered.  Only records matching exactly those
 *          files and repositories are shown.  Sorting by "module", then
 *          filename, is implied.  If -l ("last modified") is specified,
 *          then "update" records (types WUCG), tag and release records
 *          are ignored and the last (by date) "modified" record.
 *
 *   TAG history
 *
 *     -T     All Tag records are displayed.
 *
 *** Modifiers.
 *
 *   Since ...       [All records contain a timestamp, so any report
 *                    category can be limited by date.]
 *
 *     -D date       - The "date" is parsed into a Unix "time_t" and
 *                     records with an earlier time stamp are ignored.
 *     -r rev/tag    - A "rev" begins with a digit.  A "tag" does not.  If
 *                     you use this option, every file is searched for the
 *                     indicated rev/tag.
 *     -t tag        - The "tag" is searched for in the history file and no
 *                     record is displayed before the tag is found.  An
 *                     error is printed if the tag is never found.
 *     -b string     - Records are printed only back to the last reference
 *                     to the string in the "module", "file" or
 *                     "repository" fields.
 *
 *   Field Selections [Simple comparisons on existing fields.  All field
 *                     selections are repeatable.]
 *
 *     -a            - All users.
 *     -u user       - If no user is given and '-a' is not given, only
 *                     records for the user typing the command are shown.
 *                     ==> If -a or -u is not specified, just use "self".
 *
 *     -f filematch  - Only records in which the "file" field contains the
 *                     string "filematch" are considered.
 *
 *     -p repository - Only records in which the "repository" string is a
 *                     prefix of the "repos" field are considered.
 *
 *     -m modulename - Only records which contain "modulename" in the
 *                     "module" field are considered.
 *
 *
 * EXAMPLES: ("cvs history", "cvs his" or "cvs hi")
 *
 *** Checked out files for username.  (default self, e.g. "dgg")
 *      cvs hi                [equivalent to: "cvs hi -o -u dgg"]
 *      cvs hi -u user        [equivalent to: "cvs hi -o -u user"]
 *      cvs hi -o             [equivalent to: "cvs hi -o -u dgg"]
 *
 *** Committed (modified) files from the beginning of the file.
 *      cvs hi -c [-u user]
 *
 *** Committed (modified) files since Midnight, January 1, 1990:
 *      cvs hi -c -D 'Jan 1 1990' [-u user]
 *
 *** Committed (modified) files since tag "TAG" was stored in the history file:
 *      cvs hi -c -t TAG [-u user]
 *
 *** Committed (modified) files since tag "TAG" was placed on the files:
 *      cvs hi -c -r TAG [-u user]
 *
 *** Who last committed file/repository X?
 *      cvs hi -c -l -[fp] X
 *
 *** Modified files since tag/date/file/repos?
 *      cvs hi -c {-r TAG | -D Date | -b string}
 *
 *** Tag history
 *      cvs hi -T
 *
 *** History of file/repository/module X.
 *      cvs hi -[fpn] X
 *
 *** History of user "user".
 *      cvs hi -e -u user
 *
 *** Dump (eXtract) specified record types
 *      cvs hi -x [TOFWUGCMAR]
 *
 *
 * FUTURE:         J[Join], I[Import]  (Not currently implemented.)
 *
 */
```

```
#include "cvs.h"
#include "savecwd.h"

static struct hrec
{
    char *type;      /* Type of record (In history record) */
    char *user;      /* Username (In history record) */
    char *dir;       /* "Compressed" Working dir (In history record) */
    char *repos;     /* (Tag is special.) Repository (In history record) */
    char *rev;       /* Revision affected (In history record) */
    char *file;      /* Filename (In history record) */
    char *end;       /* Ptr into repository to copy at end of workdir */
    char *mod;       /* The module within which the file is contained */
    time_t date;     /* Calculated from date stored in record */
    int idx;         /* Index of record, for "stable" sort. */
} *hrec_head;


static char *fill_hrec PROTO((char *line, struct hrec * hr));
static int accept_hrec PROTO((struct hrec * hr, struct hrec * lr));
static int select_hrec PROTO((struct hrec * hr));
static int sort_order PROTO((const PTR l, const PTR r));
static int within PROTO((char *find, char *string));
static void expand_modules PROTO((void));
static void read_hrecs PROTO((char *fname));
static void report_hrecs PROTO((void));
static void save_file PROTO((char *dir, char *name, char *module));
static void save_module PROTO((char *module));
static void save_user PROTO((char *name));

#define ALL_REC_TYPES "TOEFWUCGMAR"
#define USER_INCREMENT 2
#define FILE_INCREMENT 128
#define MODULE_INCREMENT 5
#define HREC_INCREMENT 128

static short report_count;

static short extract;
static short v_checkout;
static short modified;
static short tag_report;
static short module_report;
static short working;
static short last_entry;
static short all_users;

static short user_sort;
static short repos_sort;
static short file_sort;
static short module_sort;

static short tz_local;
static time_t tz_seconds_east_of_GMT;
static char *tz_name = "+0000";

/* -r, -t, or -b options, malloc'd.  These are "" if the option in
   question is not specified or is overridden by another option.  The
   main reason for using "" rather than NULL is historical.  Together
   with since_date, these are a mutually exclusive set; one overrides the
   others.  */
static char *since_rev;
static char *since_tag;
static char *backto;
/* -D option, or 0 if not specified.  RCS format.  */
static char * since_date;

static struct hrec *last_since_tag;
static struct hrec *last_backto;

/* Record types to look for, malloc'd.  Probably could be statically
   allocated, but only if we wanted to check for duplicates more than
   we do.  */
static char *rec_types;

static int hrec_count;
static int hrec_max;

static char **user_list;  /* Ptr to array of ptrs to user names */
static int user_max;      /* Number of elements allocated */
static int user_count;    /* Number of elements used */

static struct file_list_str
{
    char *l_file;
    char *l_module;
} *file_list;             /* Ptr to array file name structs */
static int file_max;      /* Number of elements allocated */
static int file_count;    /* Number of elements used */
```

```
static char **mod_list; /* Ptr to array of ptrs to module names */
static int mod_max;     /* Number of elements allocated */
static int mod_count;   /* Number of elements used */

static char *histfile;      /* Ptr to the history file name */

/* This is pretty unclear.  First of all, separating "flags" vs.
   "options" (I think the distinction is that "options" take arguments)
   is nonstandard, and not something we do elsewhere in CVS.  Second of
   all, what does "reports" mean?  I think it means that you can only
   supply one of those options, but "reports" hardly has that meaning in
   a self-explanatory way.  */
static const char *const history_usg[] =
{
    "Usage: %s %s [-report] [-flags] [-options args] [files...]\n\n",
    "   Reports:\n",
    "       -T            Produce report on all TAGs\n",
    "       -c            Committed (Modified) files\n",
    "       -o            Checked out modules\n",
    "       -m <module>    Look for specified module (repeatable)\n",
    "       -x [TOEFWUCGMAR] Extract by record type\n",
    "   Flags:\n",
    "       -a            All users (Default is self)\n",
    "       -e            Everything (same as -x, but all record types)\n",
    "       -l            Last modified (committed or modified report)\n",
    "       -w            Working directory must match\n",
    "   Options:\n",
    "       -D <date>      Since date (Many formats)\n",
    "       -b <str>       Back to record with str in module/file/repos field\n",
    "       -f <file>      Specified file (same as command line) (repeatable)\n",
    "       -n <modulename> In module (repeatable)\n",
    "       -p <repos>     In repository (repeatable)\n",
    "       -r <rev/tag>    Since rev or tag (looks inside RCS files!)\n",
    "       -t <tag>       Since tag record placed in history file (by anyone).\n",
    "       -u <user>      For user name (repeatable)\n",
    "       -z <tz>        Output for time zone <tz> (e.g. -z -0700)\n",
    NULL};

/* Sort routine for qsort:
   - If a user is selected at all, sort it first. User-within-file is useless.
   - If a module was selected explicitly, sort next on module.
   - Then sort by file.  "File" is "repository/file" unless "working" is set,
     then it is "workdir/file".  (Revision order should always track date.)
   - Always sort timestamp last.
*/
static int
sort_order (l, r)
    const PTR l;
    const PTR r;
{
    int i;
    const struct hrec *left = (const struct hrec *) l;
    const struct hrec *right = (const struct hrec *) r;

    if (user_sort)    /* If Sort by username, compare users */
    {
        if ((i = strcmp (left->user, right->user)) != 0)
            return (i);
    }
    if (module_sort) /* If sort by modules, compare module names */
    {
        if (left->mod && right->mod)
            if ((i = strcmp (left->mod, right->mod)) != 0)
                return (i);
    }
    if (repos_sort)  /* If sort by repository, compare them. */
    {
        if ((i = strcmp (left->repos, right->repos)) != 0)
            return (i);
    }
    if (file_sort)     /* If sort by filename, compare files, NOT dirs. */
    {
        if ((i = strcmp (left->file, right->file)) != 0)
            return (i);

        if (working)
        {
            if ((i = strcmp (left->dir, right->dir)) != 0)
                return (i);

            if ((i = strcmp (left->end, right->end)) != 0)
                return (i);
        }
    }

    /*
     * By default, sort by date, time
     * XXX: This fails after 2030 when date slides into sign bit
     */
```

```
    if ((i = ((long) (left->date) - (long) (right->date))) != 0)
        return (i);

    /* For matching dates, keep the sort stable by using record index */
    return (left->idx - right->idx);
}

int
history (argc, argv)
    int argc;
    char **argv;
{
    int i, c;
    char *fname;

    if (argc == -1)
        usage (history_usg);

    since_rev = xstrdup ("");
    since_tag = xstrdup ("");
    backto = xstrdup ("");
    rec_types = xstrdup ("");
    optind = 0;
    while ((c = getopt (argc, argv, "+Tacelow?D:b:f:m:n:p:r:t:u:x:X:z:")) != -1)
    {
        switch (c)
        {
            case 'T':               /* Tag list */
                report_count++;
                tag_report++;
                break;
            case 'a':               /* For all usernames */
                all_users++;
                break;
            case 'c':
                report_count++;
                modified = 1;
                break;
            case 'e':
                report_count++;
                extract++;
                free (rec_types);
                rec_types = xstrdup (ALL_REC_TYPES);
                break;
            case 'l':               /* Find Last file record */
                last_entry = 1;
                break;
            case 'o':
                report_count++;
                v_checkout = 1;
                break;
            case 'w':               /* Match Working Dir (CurDir) fields */
                working = 1;
                break;
            case 'X':               /* Undocumented debugging flag */
                histfile = optarg;
                break;
            case 'D':               /* Since specified date */
                if (*since_rev || *since_tag || *backto)
                {
                    error (0, 0, "date overriding rev/tag/backto");
                    *since_rev = *since_tag = *backto = '\0';
                }
                since_date = Make_Date (optarg);
                break;
            case 'b':               /* Since specified file/Repos */
                if (since_date || *since_rev || *since_tag)
                {
                    error (0, 0, "backto overriding date/rev/tag");
                    *since_rev = *since_tag = '\0';
                    if (since_date != NULL)
                        free (since_date);
                    since_date = NULL;
                }
                free (backto);
                backto = xstrdup (optarg);
                break;
            case 'f':               /* For specified file */
                save_file ("", optarg, (char *) NULL);
                break;
            case 'm':               /* Full module report */
                report_count++;
                module_report++;
            case 'n':               /* Look for specified module */
                save_module (optarg);
                break;
            case 'p':               /* For specified directory */
                save_file (optarg, "", (char *) NULL);
                break;
            case 'r':               /* Since specified Tag/Rev */
```

```
            if (since_date || *since_tag || *backto)
450         {
                error (0, 0, "rev overriding date/tag/backto");
                *since_tag = *backto = '\0';
                if (since_date != NULL)
                    free (since_date);
                since_date = NULL;
            }
            free (since_rev);
            since_rev = xstrdup (optarg);
            break;
460     case 't':               /* Since specified Tag/Rev */
            if (since_date || *since_rev || *backto)
            {
                error (0, 0, "tag overriding date/marker/file/repos");
                *since_rev = *backto = '\0';
                if (since_date != NULL)
                    free (since_date);
                since_date = NULL;
            }
            free (since_tag);
470         since_tag = xstrdup (optarg);
            break;
        case 'u':               /* For specified username */
            save_user (optarg);
            break;
        case 'x':
            report_count++;
            extract++;
            {
                char *cp;
480
                for (cp = optarg; *cp; cp++)
                    if (!strchr (ALL_REC_TYPES, *cp))
                        error (1, 0, "%c is not a valid report type", *cp);
            }
            free (rec_types);
            rec_types = xstrdup (optarg);
            break;
        case 'z':
            tz_local =
490             (optarg[0] == 'l' || optarg[0] == 'L')
                && (optarg[1] == 't' || optarg[1] == 'T')
                && !optarg[2];
            if (tz_local)
                tz_name = optarg;
            else
            {
                /*
                 * Convert a known time with the given timezone to time_t.
                 * Use the epoch + 23 hours, so timezones east of GMT work.
500              */
                static char f[] = "1/1/1970 23:00 %s";
                char *buf = xmalloc (sizeof (f) - 2 + strlen (optarg));
                time_t t;
                sprintf (buf, f, optarg);
                t = get_date (buf, (struct timeb *) NULL);
                free (buf);
                if (t == (time_t) -1)
                    error (0, 0, "%s is not a known time zone", optarg);
                else
510             {
                    /*
                     * Convert to seconds east of GMT, removing the
                     * 23-hour offset mentioned above.
                     */
                    tz_seconds_east_of_GMT = (time_t)23 * 60 * 60  -  t;
                    tz_name = optarg;
                }
            }
            break;
520     case '?':
        default:
            usage (history_usg);
            break;
    }
}
c = optind;                     /* Save the handled option count */

/* ================ Now analyze the arguments a bit */
if (!report_count)
530     v_checkout++;
else if (report_count > 1)
    error (1, 0, "Only one report type allowed from: \"-Tcomx\".");

#ifdef CLIENT_SUPPORT
if (client_active)
{
    struct file_list_str *f1;
    char **mod;
```

```
540         /* We're the client side.  Fire up the remote server.  */
            start_server ();

            ign_setup ();

            if (tag_report)
                send_arg("-T");
            if (all_users)
                send_arg("-a");
            if (modified)
550             send_arg("-c");
            if (last_entry)
                send_arg("-l");
            if (v_checkout)
                send_arg("-o");
            if (working)
                send_arg("-w");
            if (histfile)
                send_arg("-X");
            if (since_date)
560             client_senddate (since_date);
            if (backto[0] != '\0')
                option_with_arg ("-b", backto);
            for (f1 = file_list; f1 < &file_list[file_count]; ++f1)
            {
                if (f1->l_file[0] == '*')
                    option_with_arg ("-p", f1->l_file + 1);
                else
                    option_with_arg ("-f", f1->l_file);
            }
570         if (module_report)
                send_arg("-m");
            for (mod = mod_list; mod < &mod_list[mod_count]; ++mod)
                option_with_arg ("-n", *mod);
            if (*since_rev)
                option_with_arg ("-r", since_rev);
            if (*since_tag)
                option_with_arg ("-t", since_tag);
            for (mod = user_list; mod < &user_list[user_count]; ++mod)
                option_with_arg ("-u", *mod);
580         if (extract)
                option_with_arg ("-x", rec_types);
            option_with_arg ("-z", tz_name);

            send_to_server ("history\012", 0);
            return get_responses_and_close ();
        }
    #endif

        if (all_users)
590         save_user ("");

        if (mod_list)
            expand_modules ();

        if (tag_report)
        {
            if (!strchr (rec_types, 'T'))
            {
                rec_types = xrealloc (rec_types, strlen (rec_types) + 5);
600             (void) strcat (rec_types, "T");
            }
        }
        else if (extract)
        {
            if (user_list)
                user_sort++;
        }
        else if (modified)
        {
610         free (rec_types);
            rec_types = xstrdup ("MAR");
            /*
             * If the user has not specified a date oriented flag ("Since"), sort
             * by Repository/file before date.  Default is "just" date.
             */
            if (!since_date && !*since_rev && !*since_tag && !*backto)
            {
                repos_sort++;
                file_sort++;
620             /*
                 * If we are not looking for last_modified and the user specified
                 * one or more users to look at, sort by user before filename.
                 */
                if (!last_entry && user_list)
                    user_sort++;
            }
        }
        else if (module_report)
```

```
    {
630     free (rec_types);
        rec_types = xstrdup (last_entry ? "OMAR" : ALL_REC_TYPES);
        module_sort++;
        repos_sort++;
        file_sort++;
        working = 0;                /* User's workdir doesn't count here */
    }
    else
        /* Must be "checkout" or default */
    {
640     free (rec_types);
        rec_types = xstrdup ("OF");
        /* See comments in "modified" above */
        if (!last_entry && user_list)
            user_sort++;
        if (!since_date && !*since_rev && !*since_tag && !*backto)
            file_sort++;
    }

    /* If no users were specified, use self (-a saves a universal ("") user) */
650 if (!user_list)
        save_user (getcaller ());

    /* If we're looking back to a Tag value, must consider "Tag" records */
    if (*since_tag && !strchr (rec_types, 'T'))
    {
        rec_types = xrealloc (rec_types, strlen (rec_types) + 5);
        (void) strcat (rec_types, "T");
    }

660 argc -= c;
    argv += c;
    for (i = 0; i < argc; i++)
        save_file ("", argv[i], (char *) NULL);

    if (histfile)
        fname = xstrdup (histfile);
    else
    {
        fname = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM)
670                      + sizeof (CVSROOTADM_HISTORY) + 10);
        (void) sprintf (fname, "%s/%s/%s", CVSroot_directory,
                        CVSROOTADM, CVSROOTADM_HISTORY);
    }

    read_hrecs (fname);
    qsort ((PTR) hrec_head, hrec_count, sizeof (struct hrec), sort_order);
    report_hrecs ();
    free (fname);
    if (since_date != NULL)
680     free (since_date);
    free (since_rev);
    free (since_tag);
    free (backto);
    free (rec_types);

    return (0);
}

void
690 history_write (type, update_dir, revs, name, repository)
    int type;
    char *update_dir;
    char *revs;
    char *name;
    char *repository;
{
    char *fname;
    char *workdir;
    char *username = getcaller ();
700 int fd;
    char *line;
    char *slash = "", *cp, *cp2, *repos;
    int i;
    static char *tilde = "";
    static char *PrCurDir = NULL;

    if (logoff)                 /* History is turned off by cmd line switch */
        return;
    fname = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM)
710                  + sizeof (CVSROOTADM_HISTORY) + 10);
    (void) sprintf (fname, "%s/%s/%s", CVSroot_directory,
                    CVSROOTADM, CVSROOTADM_HISTORY);

    /* turn off history logging if the history file does not exist */
    if (!isfile (fname))
    {
        logoff = 1;
        goto out;
```

```
    }

    if (trace)
#ifdef SERVER_SUPPORT
        fprintf (stderr, "%c-> fopen(%s,a)\n",
                 (server_active) ? 'S' : ' ', fname);
#else
        fprintf (stderr, "-> fopen(%s,a)\n", fname);
#endif
    if (noexec)
        goto out;
    fd = CVS_OPEN (fname, O_WRONLY | O_APPEND | O_CREAT | OPEN_BINARY, 0666);
    if (fd < 0)
        error (1, errno, "cannot open history file: %s", fname);

    repos = Short_Repository (repository);

    if (!PrCurDir)
    {
        char *pwdir;

        pwdir = get_homedir ();
        PrCurDir = CurDir;
        if (pwdir != NULL)
        {
            /* Assumes neither CurDir nor pwdir ends in '/' */
            i = strlen (pwdir);
            if (!strncmp (CurDir, pwdir, i))
            {
                PrCurDir += i;      /* Point to '/' separator */
                tilde = "~";
            }
            else
            {
                /* Try harder to find a "homedir" */
                struct saved_cwd cwd;
                char *homedir;

                if (save_cwd (&cwd))
                    error_exit ();

                if ( CVS_CHDIR (pwdir) < 0)
                    error (1, errno, "can't chdir(%s)", pwdir);
                homedir = xgetwd ();
                if (homedir == NULL)
                    error (1, errno, "can't getwd in %s", pwdir);

                if (restore_cwd (&cwd, NULL))
                    error_exit ();
                free_cwd (&cwd);

                i = strlen (homedir);
                if (!strncmp (CurDir, homedir, i))
                {
                    PrCurDir += i; /* Point to '/' separator */
                    tilde = "~";
                }
                free (homedir);
            }
        }
    }

    if (type == 'T')
    {
        repos = update_dir;
        update_dir = "";
    }
    else if (update_dir && *update_dir)
        slash = "/";
    else
        update_dir = "";

    workdir = xmalloc (strlen (tilde) + strlen (PrCurDir) + strlen (slash)
                       + strlen (update_dir) + 10);
    (void) sprintf (workdir, "%s%s%s%s", tilde, PrCurDir, slash, update_dir);

    /*
     * "workdir" is the directory where the file "name" is. ("~" == $HOME)
     * "repos" is the Repository, relative to $CVSROOT where the RCS file is.
     *
     * "$workdir/$name" is the working file name.
     * "$CVSROOT/$repos/$name,v" is the RCS file in the Repository.
     *
     * First, note that the history format was intended to save space, not
     * to be human readable.
     *
     * The working file directory ("workdir") and the Repository ("repos")
     * usually end with the same one or more directory elements.  To avoid
     * duplication (and save space), the "workdir" field ends with
     * an integer offset into the "repos" field.  This offset indicates the
```

```
     * beginning of the "tail" of "repos", after which all characters are
     * duplicates.
     *
     * In other words, if the "workdir" field has a '*' (a very stupid thing
     * to put in a filename) in it, then every thing following the last '*'
     * is a hex offset into "repos" of the first character from "repos" to
     * append to "workdir" to finish the pathname.
     *
     * It might be easier to look at an example:
     *
     *  M273b3463|dgg|~/work*9|usr/local/cvs/examples|1.2|loginfo
     *
     * Indicates that the workdir is really "~/work/cvs/examples", saving
     * 10 characters, where "~/work*d" would save 6 characters and mean that
     * the workdir is really "~/work/examples".  It will mean more on
     * directories like: usr/local/gnu/emacs/dist-19.17/lisp/term
     *
     * "workdir" is always an absolute pathname (~/xxx is an absolute path)
     * "repos" is always a relative pathname.  So we can assume that we will
     * never run into the top of "workdir" - there will always be a '/' or
     * a '~' at the head of "workdir" that is not matched by anything in
     * "repos".  On the other hand, we *can* run off the top of "repos".
     *
     * Only "compress" if we save characters.
     */

    if (!repos)
        repos = "";

    cp = workdir + strlen (workdir) - 1;
    cp2 = repos + strlen (repos) - 1;
    for (i = 0; cp2 >= repos && cp > workdir && *cp == *cp2--; cp--)
        i++;

    if (i > 2)
    {
        i = strlen (repos) - i;
        (void) sprintf ((cp + 1), "*%x", i);
    }

    if (!revs)
        revs = "";
    line = xmalloc (strlen (username) + strlen (workdir) + strlen (repos)
                    + strlen (revs) + strlen (name) + 100);
    sprintf (line, "%c%08lx|%s|%s|%s|%s\n",
             type, (long) time ((time_t *) NULL),
             username, workdir, repos, revs, name);

    /* Lessen some race conditions on non-Posix-compliant hosts.  */
    if (lseek (fd, (off_t) 0, SEEK_END) == -1)
        error (1, errno, "cannot seek to end of history file: %s", fname);

    if (write (fd, line, strlen (line)) < 0)
        error (1, errno, "cannot write to history file: %s", fname);
    free (line);
    if (close (fd) != 0)
        error (1, errno, "cannot close history file: %s", fname);
    free (workdir);
 out:
    free (fname);
}

/*
 * save_user() adds a user name to the user list to select.  Zero-length
 *             username ("") matches any user.
 */
static void
save_user (name)
    char *name;
{
    if (user_count == user_max)
    {
        user_max += USER_INCREMENT;
        user_list = (char **) xrealloc ((char *) user_list,
                                        (int) user_max * sizeof (char *));
    }
    user_list[user_count++] = xstrdup (name);
}

/*
 * save_file() adds file name and associated module to the file list to select.
 *
 * If "dir" is null, store a file name as is.
 * If "name" is null, store a directory name with a '*' on the front.
 * Else, store concatenated "dir/name".
 *
 * Later, in the "select" stage:
 *      - if it starts with '*', it is prefix-matched against the repository.
 *      - if it has a '/' in it, it is matched against the repository/file.
 *      - else it is matched against the file name.
```

```
 */
static void
save_file (dir, name, module)
    char *dir;
    char *name;
    char *module;
{
    char *cp;
    struct file_list_str *fl;

    if (file_count == file_max)
    {
        file_max += FILE_INCREMENT;
        file_list = (struct file_list_str *) xrealloc ((char *) file_list,
                                                       file_max * sizeof (*fl));
    }
    fl = &file_list[file_count++];
    fl->l_file = cp = xmalloc (strlen (dir) + strlen (name) + 2);
    fl->l_module = module;

    if (dir && *dir)
    {
        if (name && *name)
        {
            (void) strcpy (cp, dir);
            (void) strcat (cp, "/");
            (void) strcat (cp, name);
        }
        else
        {
            *cp++ = '*';
            (void) strcpy (cp, dir);
        }
    }
    else
    {
        if (name && *name)
        {
            (void) strcpy (cp, name);
        }
        else
        {
            error (0, 0, "save_file: null dir and file name");
        }
    }
}

static void
save_module (module)
    char *module;
{
    if (mod_count == mod_max)
    {
        mod_max += MODULE_INCREMENT;
        mod_list = (char **) xrealloc ((char *) mod_list,
                                       mod_max * sizeof (char *));
    }
    mod_list[mod_count++] = xstrdup (module);
}

static void
expand_modules ()
{
}

/* fill_hrec
 *
 * Take a ptr to 7-part history line, ending with a newline, for example:
 *
 *      M273b3463|dgg|~/work*9|usr/local/cvs/examples|1.2|loginfo
 *
 * Split it into 7 parts and drop the parts into a "struct hrec".
 * Return a pointer to the character following the newline.
 */

#define NEXT_BAR(here) do { while (isspace(*line)) line++; hr->here = line; \
while ((c = *line++) && c != '|') ; if (!c) return(rtn); \
*(line - 1) = '\0'; } while (0)

static char *
fill_hrec (line, hr)
    char *line;
    struct hrec *hr;
{
    char *cp, *rtn;
    int c;
    int off;
    static int idx = 0;
    unsigned long date;
```

```
    memset ((char *) hr, 0, sizeof (*hr));
    while (isspace (*line))
        line++;
    if (!(rtn = strchr (line, '\n')))
        return ("");
    *rtn++ = '\0';

    hr->type = line++;
    (void) sscanf (line, "%lx", &date);
    hr->date = date;
    while (*line && strchr ("0123456789abcdefABCDEF", *line))
        line++;
    if (*line == '\0')
        return (rtn);

    line++;
    NEXT_BAR (user);
    NEXT_BAR (dir);
    if ((cp = strrchr (hr->dir, '*')) != NULL)
    {
        *cp++ = '\0';
        (void) sscanf (cp, "%x", &off);
        hr->end = line + off;
    }
    else
        hr->end = line - 1;       /* A handy pointer to '\0' */
    NEXT_BAR (repos);
    NEXT_BAR (rev);
    hr->idx = idx++;
    if (strchr ("FOET", *(hr->type)))
        hr->mod = line;

    NEXT_BAR (file); /* This returns ptr to next line or final '\0' */
    return (rtn);    /* If it falls through, go on to next record */
}

/* read_hrecs's job is to read the history file and fill in all the "hrec"
 * (history record) array elements with the ones we need to print.
 *
 * Logic:
 * - Read the whole history file into a single buffer.
 * - Walk through the buffer, parsing lines out of the buffer.
 *   1. Split line into pointer and integer fields in the "next" hrec.
 *   2. Apply tests to the hrec to see if it is wanted.
 *   3. If it *is* wanted, bump the hrec pointer down by one.
 */
static void
read_hrecs (fname)
    char *fname;
{
    char *cp, *cp2;
    int i, fd;
    struct hrec *hr;
    struct stat st_buf;

    if ((fd = CVS_OPEN (fname, O_RDONLY | OPEN_BINARY)) < 0)
        error (1, errno, "cannot open history file: %s", fname);

    if (fstat (fd, &st_buf) < 0)
        error (1, errno, "can't stat history file");

    /* Exactly enough space for lines data */
    if (!(i = st_buf.st_size))
        error (1, 0, "history file is empty");
    cp = xmalloc (i + 2);

    if (read (fd, cp, i) != i)
        error (1, errno, "cannot read log file");
    (void) close (fd);

    if (*(cp + i - 1) != '\n')
    {
        *(cp + i) = '\n';           /* Make sure last line ends in '\n' */
        i++;
    }
    *(cp + i) = '\0';
    for (cp2 = cp; cp2 - cp < i; cp2++)
    {
        if (*cp2 != '\n' && !isprint (*cp2))
            *cp2 = ' ';
    }

    hrec_max = HREC_INCREMENT;
    hrec_head = (struct hrec *) xmalloc (hrec_max * sizeof (struct hrec));

    while (*cp)
    {
        if (hrec_count == hrec_max)
        {
            struct hrec *old_head = hrec_head;
```

```
            hrec_max += HREC_INCREMENT;
            hrec_head = (struct hrec *) xrealloc ((char *) hrec_head,
                                                  hrec_max * sizeof (struct hrec));
            if (hrec_head != old_head)
            {
                if (last_since_tag)
                    last_since_tag = hrec_head + (last_since_tag - old_head);
                if (last_backto)
                    last_backto = hrec_head + (last_backto - old_head);
            }
        }

        hr = hrec_head + hrec_count;
        cp = fill_hrec (cp, hr); /* cp == next line or '\0' at end of buffer */

        if (select_hrec (hr))
            hrec_count++;
    }

    /* Special selection problem: If "since_tag" is set, we have saved every
     * record from the 1st occurrence of "since_tag", when we want to save
     * records since the *last* occurrence of "since_tag".  So what we have
     * to do is bump hrec_head forward and reduce hrec_count accordingly.
     */
    if (last_since_tag)
    {
        hrec_count -= (last_since_tag - hrec_head);
        hrec_head = last_since_tag;
    }

    /* Much the same thing is necessary for the "backto" option. */
    if (last_backto)
    {
        hrec_count -= (last_backto - hrec_head);
        hrec_head = last_backto;
    }
}

/* Utility program for determining whether "find" is inside "string" */
static int
within (find, string)
    char *find, *string;
{
    int c, len;

    if (!find || !string)
        return (0);

    c = *find++;
    len = strlen (find);

    while (*string)
    {
        if (!(string = strchr (string, c)))
            return (0);
        string++;
        if (!strncmp (find, string, len))
            return (1);
    }
    return (0);
}

/* The purpose of "select_hrec" is to apply the selection criteria based on
 * the command arguments and defaults and return a flag indicating whether
 * this record should be remembered for printing.
 */
static int
select_hrec (hr)
    struct hrec *hr;
{
    char **cpp, *cp, *cp2;
    struct file_list_str *fl;
    int count;

    /* "Since" checking:  The argument parser guarantees that only one of the
     *                    following four choices is set:
     *
     * 1. If "since_date" is set, it contains the date specified on the
     *    command line. hr->date fields earlier than "since_date" are ignored.
     * 2. If "since_rev" is set, it contains either an RCS "dotted" revision
     *    number (which is of limited use) or a symbolic TAG.  Each RCS file
     *    is examined and the date on the specified revision (or the revision
     *    corresponding to the TAG) in the RCS file (CVSROOT/repos/file) is
     *    compared against hr->date as in 1. above.
     * 3. If "since_tag" is set, matching tag records are saved.  The field
     *    "last_since_tag" is set to the last one of these.  Since we don't
     *    know where the last one will be, all records are saved from the
     *    first occurrence of the TAG.  Later, at the end of "select_hrec"
     *    records before the last occurrence of "since_tag" are skipped.
```

```
     * 4. If "backto" is set, all records with a module name or file name
1170 *    matching "backto" are saved.  In addition, all records with a
     *    repository field with a *prefix* matching "backto" are saved.
     *    The field "last_backto" is set to the last one of these.  As in
     *    3. above, "select_hrec" adjusts to include the last one later on.
     */
    if (since_date)
    {
        char *ourdate = date_from_time_t (hr->date);

        if (RCS_datecmp (ourdate, since_date) < 0)
1180        return (0);

        free (ourdate);
    }
    else if (*since_rev)
    {
        Vers_TS *vers;
        time_t t;
        struct file_info finfo;

1190    memset (&finfo, 0, sizeof finfo);
        finfo.file = hr->file;
        /* Not used, so don't worry about it.  */
        finfo.update_dir = NULL;
        finfo.fullname = finfo.file;
        finfo.repository = hr->repos;
        finfo.entries = NULL;
        finfo.rcs = NULL;

        vers = Version_TS (&finfo, (char *) NULL, since_rev, (char *) NULL,
1200                       1, 0);
        if (vers->vn_rcs)
        {
            if ((t = RCS_getrevtime (vers->srcfile, vers->vn_rcs, (char *) 0, 0))
                != (time_t) 0)
            {
                if (hr->date < t)
                {
                    freevers_ts (&vers);
                    return (0);
1210            }
            }
        }
        freevers_ts (&vers);
    }
    else if (*since_tag)
    {
        if (*(hr->type) == 'T')
        {
            /*
1220         * A 'T'ag record, the "rev" field holds the tag to be set,
             * while the "repos" field holds "D"elete, "A"dd or a rev.
             */
            if (within (since_tag, hr->rev))
            {
                last_since_tag = hr;
                return (1);
            }
            else
                return (0);
1230    }
        if (!last_since_tag)
            return (0);
    }
    else if (*backto)
    {
        if (within (backto, hr->file) || within (backto, hr->mod) ||
            within (backto, hr->repos))
            last_backto = hr;
        else
1240        return (0);
    }

    /* User checking:
     *
     * Run down "user_list", match username ("" matches anything)
     * If "" is not there and actual username is not there, return failure.
     */
    if (user_list && hr->user)
    {
1250    for (cpp = user_list, count = user_count; count; cpp++, count--)
        {
            if (!**cpp)
                break;              /* null user == accept */
            if (!strcmp (hr->user, *cpp)) /* found listed user */
                break;
        }
        if (!count)
            return (0);             /* Not this user */
```

```
    }

    /* Record type checking:
     *
     * 1. If Record type is not in rec_types field, skip it.
     * 2. If mod_list is null, keep everything.  Otherwise keep only modules
     *    on mod_list.
     * 3. If neither a 'T', 'F' nor 'O' record, run through "file_list".  If
     *    file_list is null, keep everything.  Otherwise, keep only files on
     *    file_list, matched appropriately.
     */
    if (!strchr (rec_types, *(hr->type)))
        return (0);
    if (!strchr ("TFOE", *(hr->type))) /* Don't bother with "file" if "TFOE" */
    {
        if (file_list)                  /* If file_list is null, accept all */
        {
            for (fl = file_list, count = file_count; count; fl++, count--)
            {
                /* 1. If file_list entry starts with '*', skip the '*' and
                 *    compare it against the repository in the hrec.
                 * 2. If file_list entry has a '/' in it, compare it against
                 *    the concatenation of the repository and file from hrec.
                 * 3. Else compare the file_list entry against the hrec file.
                 */
                char *cmpfile = NULL;

                if (*(cp = fl->l_file) == '*')
                {
                    cp++;
                    /* if argument to -p is a prefix of repository */
                    if (!strncmp (cp, hr->repos, strlen (cp)))
                    {
                        hr->mod = fl->l_module;
                        break;
                    }
                }
                else
                {
                    if (strchr (cp, '/'))
                    {
                        cmpfile = xmalloc (strlen (hr->repos)
                                           + strlen (hr->file)
                                           + 10);
                        (void) sprintf (cmpfile, "%s/%s",
                                        hr->repos, hr->file);
                        cp2 = cmpfile;
                    }
                    else
                    {
                        cp2 = hr->file;
                    }

                    /* if requested file is found within {repos}/file fields */
                    if (within (cp, cp2))
                    {
                        hr->mod = fl->l_module;
                        break;
                    }
                    if (cmpfile != NULL)
                        free (cmpfile);
                }
            }
            if (!count)
                return (0);          /* String specified and no match */
        }
    }
    if (mod_list)
    {
        for (cpp = mod_list, count = mod_count; count; cpp++, count--)
        {
            if (hr->mod && !strcmp (hr->mod, *cpp)) /* found module */
                break;
        }
        if (!count)
            return (0); /* Module specified & this record is not one of them. */
    }

    return (1);       /* Select this record unless rejected above. */
}

/* The "sort_order" routine (when handed to qsort) has arranged for the
 * hrecs files to be in the right order for the report.
 *
 * Most of the "selections" are done in the select_hrec routine, but some
 * selections are more easily done after the qsort by "accept_hrec".
 */
static void
report_hrecs ()
{
```

```
    struct hrec *hr, *lr;
1350 struct tm *tm;
    int i, count, ty;
    char *cp;
    int user_len, file_len, rev_len, mod_len, repos_len;

    if (*since_tag && !last_since_tag)
    {
        (void) printf ("No tag found: %s\n", since_tag);
        return;
    }
1360 else if (*backto && !last_backto)
    {
        (void) printf ("No module, file or repository with: %s\n", backto);
        return;
    }
    else if (hrec_count < 1)
    {
        (void) printf ("No records selected.\n");
        return;
    }
1370
    user_len = file_len = rev_len = mod_len = repos_len = 0;

    /* Run through lists and find maximum field widths */
    hr = lr = hrec_head;
    hr++;
    for (count = hrec_count; count--; lr = hr, hr++)
    {
        char *repos;

1380    if (!count)
            hr = NULL;
        if (!accept_hrec (lr, hr))
            continue;

        ty = *(lr->type);
        repos = xstrdup (lr->repos);
        if ((cp = strrchr (repos, '/')) != NULL)
        {
            if (lr->mod && !strcmp (++cp, lr->mod))
1390        {
                (void) strcpy (cp, "*");
            }
        }
        if ((i = strlen (lr->user)) > user_len)
            user_len = i;
        if ((i = strlen (lr->file)) > file_len)
            file_len = i;
        if (ty != 'T' && (i = strlen (repos)) > repos_len)
            repos_len = i;
1400    if (ty != 'T' && (i = strlen (lr->rev)) > rev_len)
            rev_len = i;
        if (lr->mod && (i = strlen (lr->mod)) > mod_len)
            mod_len = i;
        free (repos);
    }

    /* Walk through hrec array setting "lr" (Last Record) to each element.
     * "hr" points to the record following "lr" - It is NULL in the last
     * pass.
1410 *
     * There are two sections in the loop below:
     * 1. Based on the report type (e.g. extract, checkout, tag, etc.),
     *    decide whether the record should be printed.
     * 2. Based on the record type, format and print the data.
     */
    for (lr = hrec_head, hr = (lr + 1); hrec_count--; lr = hr, hr++)
    {
        char *workdir;
        char *repos;
1420
        if (!hrec_count)
            hr = NULL;
        if (!accept_hrec (lr, hr))
            continue;

        ty = *(lr->type);
        if (!tz_local)
        {
            time_t t = lr->date + tz_seconds_east_of_GMT;
1430        tm = gmtime (&t);
        }
        else
            tm = localtime (&(lr->date));

        (void) printf ("%c %02d/%02d %02d:%02d %s %-*s", ty, tm->tm_mon + 1,
                tm->tm_mday, tm->tm_hour, tm->tm_min, tz_name,
                user_len, lr->user);
```

```
        workdir = xmalloc (strlen (lr->dir) + strlen (lr->end) + 10);
1440    (void) sprintf (workdir, "%s%s", lr->dir, lr->end);
        if ((cp = strrchr (workdir, '/')) != NULL)
        {
            if (lr->mod && !strcmp (++cp, lr->mod))
            {
                (void) strcpy (cp, "*");
            }
        }
        repos = xmalloc (strlen (lr->repos) + 10);
        (void) strcpy (repos, lr->repos);
1450    if ((cp = strrchr (repos, '/')) != NULL)
        {
            if (lr->mod && !strcmp (++cp, lr->mod))
            {
                (void) strcpy (cp, "*");
            }
        }

        switch (ty)
        {
1460        case 'T':
            /* 'T'ag records: repository is a "tag type", rev is the tag */
            (void) printf (" %-*s [%s:%s]", mod_len, lr->mod, lr->rev,
                           repos);
            if (working)
                (void) printf (" {%s}", workdir);
            break;
        case 'F':
        case 'E':
        case 'O':
1470        if (lr->rev && *(lr->rev))
                (void) printf (" [%s]", lr->rev);
            (void) printf (" %-*s =%s%-*s %s", repos_len, repos, lr->mod,
                           mod_len + 1 - (int) strlen (lr->mod),
                           "=", workdir);
            break;
        case 'W':
        case 'U':
        case 'C':
        case 'G':
1480    case 'M':
        case 'A':
        case 'R':
            (void) printf (" %-*s %-*s %-*s =%s= %s", rev_len, lr->rev,
                           file_len, lr->file, repos_len, repos,
                           lr->mod ? lr->mod : "", workdir);
            break;
        default:
            (void) printf ("Hey! What is this junk? RecType[0x%2.2x]", ty);
            break;
1490    }
        (void) putchar ('\n');
        free (workdir);
        free (repos);
    }
}

static int
accept_hrec (lr, hr)
    struct hrec *hr, *lr;
1500 {
    int ty;

    ty = *(lr->type);

    if (last_since_tag && ty == 'T')
        return (1);

    if (v_checkout)
    {
1510    if (ty != 'O')
            return (0);             /* Only interested in 'O' records */

        /* We want to identify all the states that cause the next record
         * ("hr") to be different from the current one ("lr") and only
         * print a line at the allowed boundaries.
         */

        if (!hr ||                  /* The last record */
            strcmp (hr->user, lr->user) || /* User has changed */
1520        strcmp (hr->mod, lr->mod) ||/* Module has changed */
            (working &&             /* If must match "workdir" */
             (strcmp (hr->dir, lr->dir) || /*     and the 1st parts or */
              strcmp (hr->end, lr->end)))) /*     the 2nd parts differ */

            return (1);
    }
    else if (modified)
    {
```

```
        if (!last_entry ||          /* Don't want only last rec */
1530        !hr ||                  /* Last entry is a "last entry" */
            strcmp (hr->repos, lr->repos) || /* Repository has changed */
            strcmp (hr->file, lr->file))/* File has changed */
            return (1);

        if (working)
        {                           /* If must match "workdir" */
            if (strcmp (hr->dir, lr->dir) || /*     and the 1st parts or */
                strcmp (hr->end, lr->end)) /*     the 2nd parts differ */
                return (1);
1540    }
    }
    else if (module_report)
    {
        if (!last_entry ||          /* Don't want only last rec */
            !hr ||                  /* Last entry is a "last entry" */
            strcmp (hr->mod, lr->mod) ||/* Module has changed */
            strcmp (hr->repos, lr->repos) || /* Repository has changed */
            strcmp (hr->file, lr->file))/* File has changed */
            return (1);
1550    }
    else
    {
        /* "extract" and "tag_report" always print selected records. */
        return (1);
    }

    return (0);
}
```

A.30 ignore.c

```
/* This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

/*
 * .cvsignore file support contributed by David G. Grubbs <dggodi.com>
*/

#include "cvs.h"
#include "getline.h"

/*
 * Ignore file section.
 *
 *      "!" may be included any time to reset the list (i.e. ignore nothing);
 *      "*" may be specified to ignore everything.  It stays as the first
 *          element forever, unless a "!" clears it out.
 */

static char **ign_list;          /* List of files to ignore in update
                                  * and import */
static char **s_ign_list = NULL;
static int ign_count;            /* Number of active entries */
static int s_ign_count = 0;
static int ign_size;             /* This many slots available (plus
                                  * one for a NULL) */
static int ign_hold = -1;        /* Index where first "temporary" item
                                  * is held */

const char *ign_default = ". .. core RCSLOG tags TAGS RCS SCCS .make.state\
 .nse_depinfo #* .#* cvslog.* ,* CVS CVS.adm .del-* *.a *.olb *.o *.obj\
 *.so *.Z *~ *.old *.elc *.ln *.bak *.BAK *.orig *.rej *.exe _$* *$";

#define IGN_GROW 16              /* grow the list by 16 elements at a
                                  * time */

/* Nonzero if we have encountered an -I ! directive, which means one should
   no longer ask the server about what is in CVSROOTADM_IGNORE.  */
int ign_inhibit_server;

/*
 * To the "ignore list", add the hard-coded default ignored wildcards above,
 * the wildcards found in $CVSROOT/CVSROOT/cvsignore, the wildcards found in
 * ~/.cvsignore and the wildcards found in the CVSIGNORE environment
 * variable.
 */
void
ign_setup ()
{
    char *home_dir;
    char *tmp;

    ign_inhibit_server = 0;

    /* Start with default list and special case */
    tmp = xstrdup (ign_default);
    ign_add (tmp, 0);
    free (tmp);

#ifdef CLIENT_SUPPORT
    /* The client handles another way, by (after it does its own ignore file
       processing, and only if !ign_inhibit_server), letting the server
       know about the files and letting it decide whether to ignore
       them based on CVSROOOTADM_IGNORE.  */
    if (!client_active)
#endif
    {
        char *file = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM)
                              + sizeof (CVSROOTADM_IGNORE) + 10);
        /* Then add entries found in repository, if it exists */
        (void) sprintf (file, "%s/%s/%s", CVSroot_directory,
                        CVSROOTADM, CVSROOTADM_IGNORE);
        ign_add_file (file, 0);
        free (file);
    }

    /* Then add entries found in home dir, (if user has one) and file exists */
    home_dir = get_homedir ();
    if (home_dir)
    {
        char *file = xmalloc (strlen (home_dir) + sizeof (CVSDOTIGNORE) + 10);
        (void) sprintf (file, "%s/%s", home_dir, CVSDOTIGNORE);
```

```
        ign_add_file (file, 0);
        free (file);
    }

    /* Then add entries found in CVSIGNORE environment variable. */
    ign_add (getenv (IGNORE_ENV), 0);

    /* Later, add ignore entries found in -I arguments */
}

/*
 * Open a file and read lines, feeding each line to a line parser. Arrange
 * for keeping a temporary list of wildcards at the end, if the "hold"
 * argument is set.
 */
void
ign_add_file (file, hold)
    char *file;
    int hold;
{
    FILE *fp;
    char *line = NULL;
    size_t line_allocated = 0;

    /* restore the saved list (if any) */
    if (s_ign_list != NULL)
    {
        int i;

        for (i = 0; i < s_ign_count; i++)
            ign_list[i] = s_ign_list[i];
        ign_count = s_ign_count;
        ign_list[ign_count] = NULL;

        s_ign_count = 0;
        free (s_ign_list);
        s_ign_list = NULL;
    }

    /* is this a temporary ignore file? */
    if (hold)
    {
        /* re-set if we had already done a temporary file */
        if (ign_hold >= 0)
        {
            int i;

            for (i = ign_hold; i < ign_count; i++)
                free (ign_list[i]);
            ign_count = ign_hold;
            ign_list[ign_count] = NULL;
        }
        else
        {
            ign_hold = ign_count;
        }
    }

    /* load the file */
    fp = CVS_FOPEN (file, "r");
    if (fp == NULL)
    {
        if (! existence_error (errno))
            error (0, errno, "cannot open %s", file);
        return;
    }
    while (getline (&line, &line_allocated, fp) >= 0)
        ign_add (line, hold);
    if (ferror (fp))
        error (0, errno, "cannot read %s", file);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", file);
    free (line);
}

/* Parse a line of space-separated wildcards and add them to the list. */
void
ign_add (ign, hold)
    char *ign;
    int hold;
{
    if (!ign || !*ign)
        return;

    for (; *ign; ign++)
    {
        char *mark;
        char save;

        /* ignore whitespace before the token */
```

```
    if (isspace (*ign))
        continue;

    /*
     * if we find a single character !, we must re-set the ignore list
     * (saving it if necessary).  We also catch * as a special case in a
     * global ignore file as an optimization
     */
    if ((!*(ign+1) || isspace (*(ign+1))) && (*ign == '!' || *ign == '*'))
    {
        if (!hold)
        {
            /* permanently reset the ignore list */
            int i;

            for (i = 0; i < ign_count; i++)
                free (ign_list[i]);
            ign_count = 0;
            ign_list[0] = NULL;

            /* if we are doing a '!', continue; otherwise add the '*' */
            if (*ign == '!')
            {
                ign_inhibit_server = 1;
                continue;
            }
        }
        else if (*ign == '!')
        {
            /* temporarily reset the ignore list */
            int i;

            if (ign_hold >= 0)
            {
                for (i = ign_hold; i < ign_count; i++)
                    free (ign_list[i]);
                ign_hold = -1;
            }
            s_ign_list = (char **) xmalloc (ign_count * sizeof (char *));
            for (i = 0; i < ign_count; i++)
                s_ign_list[i] = ign_list[i];
            s_ign_count = ign_count;
            ign_count = 0;
            ign_list[0] = NULL;
            continue;
        }
    }

    /* If we have used up all the space, add some more */
    if (ign_count >= ign_size)
    {
        ign_size += IGN_GROW;
        ign_list = (char **) xrealloc ((char *) ign_list,
                                       (ign_size + 1) * sizeof (char *));
    }

    /* find the end of this token */
    for (mark = ign; *mark && !isspace (*mark); mark++)
        /* do nothing */ ;

    save = *mark;
    *mark = '\0';

    ign_list[ign_count++] = xstrdup (ign);
    ign_list[ign_count] = NULL;

    *mark = save;
    if (save)
        ign = mark;
    else
        ign = mark - 1;
    }
}

/* Set to 1 if filenames should be matched in a case-insensitive
   fashion.  Note that, contrary to the name and placement in ignore.c,
   this is no longer just for ignore patterns.  */
int ign_case;

/* Return 1 if the given filename should be ignored by update or import. */
int
ign_name (name)
    char *name;
{
    char **cpp = ign_list;

    if (cpp == NULL)
        return (0);

    if (ign_case)
```

```
    {
270     /* We do a case-insensitive match by calling fnmatch on copies of
           the pattern and the name which have been converted to
           lowercase.  FIXME: would be much cleaner to just unify this
           with the other case-insensitive fnmatch stuff (FOLD_FN_CHAR
           in lib/fnmatch.c; os2_fnmatch in emx/system.c).  */
        char *name_lower;
        char *pat_lower;
        char *p;

        name_lower = xstrdup (name);
280     for (p = name_lower; *p != '\0'; ++p)
            *p = tolower (*p);
        while (*cpp)
        {
            pat_lower = xstrdup (*cpp++);
            for (p = pat_lower; *p != '\0'; ++p)
                *p = tolower (*p);
            if (CVS_FNMATCH (pat_lower, name_lower, 0) == 0)
                goto matched;
            free (pat_lower);
290     }
        free (name_lower);
        return 0;
      matched:
        free (name_lower);
        free (pat_lower);
        return 1;
    }
    else
    {
300     while (*cpp)
            if (CVS_FNMATCH (*cpp++, name, 0) == 0)
                return 1;
        return 0;
    }
}

/* FIXME: This list of dirs to ignore stuff seems not to be used.
   Really?  send_dirent_proc and update_dirent_proc both call
   ignore_directory and do_module calls ign_dir_add.  No doubt could
310 use some documentation/testsuite work.  */

static char **dir_ign_list = NULL;
static int dir_ign_max = 0;
static int dir_ign_current = 0;

/* Add a directory to list of dirs to ignore.  */
void
ign_dir_add (name)
    char *name;
320 {
    /* Make sure we've got the space for the entry.  */
    if (dir_ign_current <= dir_ign_max)
    {
        dir_ign_max += IGN_GROW;
        dir_ign_list =
            (char **) xrealloc (dir_ign_list,
                                (dir_ign_max + 1) * sizeof (char *));
    }

330 dir_ign_list[dir_ign_current] = name;

    dir_ign_current += 1 ;
}


/* Return nonzero if NAME is part of the list of directories to ignore.  */

int
ignore_directory (name)
340 char *name;
{
    int i;

    if (!dir_ign_list)
        return 0;

    i = dir_ign_current;
    while (i--)
    {
350     if (strncmp (name, dir_ign_list[i], strlen (dir_ign_list[i])) == 0)
            return 1;
    }

    return 0;
}

/*
 * Process the current directory, looking for files not in ILIST and
```

```
 * not on the global ignore list for this directory.  If we find one,
 * call PROC passing it the name of the file and the update dir.
 * ENTRIES is the entries list, which is used to identify known
 * directories.  ENTRIES may be NULL, in which case we assume that any
 * directory with a CVS administration directory is known.
 */
void
ignore_files (ilist, entries, update_dir, proc)
    List *ilist;
    List *entries;
    char *update_dir;
    Ignore_proc proc;
{
    int subdirs;
    DIR *dirp;
    struct dirent *dp;
    struct stat sb;
    char *file;
    char *xdir;

    /* Set SUBDIRS if we have subdirectory information in ENTRIES.  */
    if (entries == NULL)
        subdirs = 0;
    else
    {
        struct stickydirtag *sdtp;

        sdtp = (struct stickydirtag *) entries->list->data;
        subdirs = sdtp == NULL || sdtp->subdirs;
    }

    /* we get called with update_dir set to "." sometimes... strip it */
    if (strcmp (update_dir, ".") == 0)
        xdir = "";
    else
        xdir = update_dir;

    dirp = CVS_OPENDIR (".");
    if (dirp == NULL)
        return;

    ign_add_file (CVSDOTIGNORE, 1);
    wrap_add_file (CVSDOTWRAPPER, 1);

    while ((dp = readdir (dirp)) != NULL)
    {
        file = dp->d_name;
        if (strcmp (file, ".") == 0 || strcmp (file, "..") == 0)
            continue;
        if (findnode_fn (ilist, file) != NULL)
            continue;
        if (subdirs)
        {
            Node *node;

            node = findnode_fn (entries, file);
            if (node != NULL
                && ((Entnode *) node->data)->type == ENT_SUBDIR)
            {
                char *p;
                int dir;

                /* For consistency with past behaviour, we only ignore
                   this directory if there is a CVS subdirectory.
                   This will normally be the case, but the user may
                   have messed up the working directory somehow.  */
                p = xmalloc (strlen (file) + sizeof CVSADM + 10);
                sprintf (p, "%s/%s", file, CVSADM);
                dir = isdir (p);
                free (p);
                if (dir)
                    continue;
            }
        }

        /* We could be ignoring FIFOs and other files which are neither
           regular files nor directories here.  */
        if (ign_name (file))
            continue;

        if (
#ifdef DT_DIR
                dp->d_type != DT_UNKNOWN ||
#endif
                lstat(file, &sb) != -1)
        {

            if (
#ifdef DT_DIR
                dp->d_type == DT_DIR || dp->d_type == DT_UNKNOWN &&
```

```
#endif
            S_ISDIR(sb.st_mode))
        {
            if (! subdirs)
            {
                char *temp;

                temp = xmalloc (strlen (file) + sizeof (CVSADM) + 10);
                (void) sprintf (temp, "%s/%s", file, CVSADM);
                if (isdir (temp))
                {
                    free (temp);
                    continue;
                }
                free (temp);
            }
        }
#ifdef S_ISLNK
        else if (
#ifdef DT_DIR
            dp->d_type == DT_LNK || dp->d_type == DT_UNKNOWN &&
#endif
            S_ISLNK(sb.st_mode))
        {
            continue;
        }
#endif
    }

    (*proc) (file, xdir);
  }
  (void) closedir (dirp);
}
```

A.31 import.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * "import" checks in the vendor release located in the current directory into
 * the CVS source repository.  The CVS vendor branch support is utilized.
 *
 * At least three arguments are expected to follow the options:
 *      repository     Where the source belongs relative to the CVSROOT
 *      VendorTag    Vendor's major tag
 *      VendorReleTag Tag for this particular release
 *
 * Additional arguments specify more Vendor Release Tags.
 */

#include "cvs.h"
#include "savecwd.h"
#include <assert.h>

static char *get_comment PROTO((char *user));
static int add_rev PROTO((char *message, RCSNode *rcs, char *vfile,
                          char *vers));
static int add_tags PROTO((RCSNode *rcs, char *vfile, char *vtag, int targc,
                           char *targv[]));
static int import_descend PROTO((char *message, char *vtag, int targc, char *targv[]));
static int import_descend_dir PROTO((char *message, char *dir, char *vtag,
                                     int targc, char *targv[]));
static int process_import_file PROTO((char *message, char *vfile, char *vtag,
                                      int targc, char *targv[]));
static int update_rcs_file PROTO((char *message, char *vfile, char *vtag, int targc,
                                  char *targv[], int inattic));
static void add_log PROTO((int ch, char *fname));

static int repos_len;
static char *vhead;
static char *vbranch;
static FILE *logfp;
static char *repository;
static int conflicts;
static int use_file_modtime;
static char *keyword_opt = NULL;

static const char *const import_usage[] =
{
    "Usage: %s %s [-d] [-k subst] [-I ign] [-m msg] [-b branch]\n",
    "    [-W spec] repository vendor-tag release-tags. . .\n",
    "\t-d\tUse the file's modification time as the time of import.\n",
    "\t-k sub\tSet default RCS keyword substitution mode.\n",
    "\t-I ign\tMore files to ignore (! to reset).\n",
    "\t-b bra\tVendor branch id.\n",
    "\t-m msg\tLog message.\n",
    "\t-W spec\tWrappers specification line.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
import (argc, argv)
    int argc;
    char **argv;
{
    char *message = NULL;
    char *tmpfile;
    char *cp;
    int i, c, msglen, err;
    List *ulist;
    Node *p;
    struct logfile_info *li;

    if (argc == -1)
        usage (import_usage);

    ign_setup ();
    wrap_setup ();

    vbranch = xstrdup (CVSBRANCH);
    optind = 0;
    while ((c = getopt (argc, argv, "+Qqdb:m:I:k:W:")) != -1)
    {
        switch (c)
        {
            case 'Q':
            case 'q':
#ifdef SERVER_SUPPORT
                /* The CVS 1.5 client sends these options (in addition to
```

```
                Global_option requests), so we must ignore them.  */
90              if (!server_active)
#endif
                    error (1, 0,
                           "-q or -Q must be specified before \"%s\"",
                           command_name);
                break;
            case 'd':
                use_file_modtime = 1;
                break;
            case 'b':
100             free (vbranch);
                vbranch = xstrdup (optarg);
                break;
            case 'm':
#ifdef FORCE_USE_EDITOR
                use_editor = 1;
#else
                use_editor = 0;
#endif
                message = xstrdup(optarg);
110             break;
            case 'I':
                ign_add (optarg, 0);
                break;
            case 'k':
                /* RCS_check_kflag returns strings of the form -kxx.  We
                   only use it for validation, so we can free the value
                   as soon as it is returned. */
                free (RCS_check_kflag (optarg));
                keyword_opt = optarg;
120             break;
            case 'W':
                wrap_add (optarg, 0);
                break;
            case '?':
            default:
                usage (import_usage);
                break;
        }
    }
130 argc -= optind;
    argv += optind;
    if (argc < 3)
        usage (import_usage);

    for (i = 1; i < argc; i++)      /* check the tags for validity */
    {
        int j;

        RCS_check_tag (argv[i]);
140     for (j = 1; j < i; j++)
            if (strcmp (argv[j], argv[i]) == 0)
                error (1, 0, "tag `%s' was specified more than once", argv[i]);
    }

    /* XXX - this should be a module, not just a pathname */
    if (! isabsolute (argv[0]))
    {
        if (CVSroot_directory == NULL)
        {
150         error (0, 0, "missing CVSROOT environment variable\n");
            error (1, 0, "Set it or specify the '-d' option to %s.",
                   program_name);
        }
        repository = xmalloc (strlen (CVSroot_directory) + strlen (argv[0])
                              + 10);
        (void) sprintf (repository, "%s/%s", CVSroot_directory, argv[0]);
        repos_len = strlen (CVSroot_directory);
    }
    else
160 {
        repository = xmalloc (strlen (argv[0]) + 5);
        (void) strcpy (repository, argv[0]);
        repos_len = 0;
    }

    /*
     * Consistency checks on the specified vendor branch.  It must be
     * composed of only numbers and dots ('.').  Also, for now we only
     * support branching to a single level, so the specified vendor branch
170  * must only have two dots in it (like "1.1.1").
     */
    for (cp = vbranch; *cp != '\0'; cp++)
        if (!isdigit (*cp) && *cp != '.')
            error (1, 0, "%s is not a numeric branch", vbranch);
    if (numdots (vbranch) != 2)
        error (1, 0, "Only branches with two dots are supported: %s", vbranch);
    vhead = xstrdup (vbranch);
    cp = strrchr (vhead, '.');
```

```
    *cp = '\0';

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        /* For rationale behind calling start_server before do_editor, see
           commit.c */
        start_server ();
    }
#endif

    if (use_editor)
    {
        do_editor ((char *) NULL, &message, repository,
                   (List *) NULL);
    }
    do_verify (message, repository);
    msglen = message == NULL ? 0 : strlen (message);
    if (msglen == 0 || message[msglen - 1] != '\n')
    {
        char *nm = xmalloc (msglen + 2);
        *nm = '\0';
        if (message != NULL)
        {
            (void) strcpy (nm, message);
            free (message);
        }
        (void) strcat (nm + msglen, "\n");
        message = nm;
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        int err;

        if (use_file_modtime)
            send_arg("-d");

        if (vbranch[0] != '\0')
            option_with_arg ("-b", vbranch);
        if (message)
            option_with_arg ("-m", message);
        if (keyword_opt != NULL)
            option_with_arg ("-k", keyword_opt);
        /* The only ignore processing which takes place on the server side
           is the CVSROOT/cvsignore file.  But if the user specified -I !,
           the documented behavior is to not process said file. */
        if (ign_inhibit_server)
        {
            send_arg ("-I");
            send_arg ("!");
        }
        wrap_send ();

        {
            int i;
            for (i = 0; i < argc; ++i)
                send_arg (argv[i]);
        }

        logfp = stdin;
        client_import_setup (repository);
        err = import_descend (message, argv[1], argc - 2, argv + 2);
        client_import_done ();
        send_to_server ("import\012", 0);
        err += get_responses_and_close ();
        return err;
    }
#endif

    /*
     * Make all newly created directories writable.  Should really use a more
     * sophisticated security mechanism here.
     */
    (void) umask (cvsumask);
    make_directories (repository);

    /* Create the logfile that will be logged upon completion */
    tmpfile = cvs_temp_name ();
    if ((logfp = CVS_FOPEN (tmpfile, "w+")) == NULL)
        error (1, errno, "cannot create temporary file `%s'", tmpfile);
    /* On systems where we can unlink an open file, do so, so it will go
       away no matter how we exit.  FIXME-maybe: Should be checking for
       errors but I'm not sure which error(s) we get if we are on a system
       where one can't unlink open files. */
    (void) CVS_UNLINK (tmpfile);
    (void) fprintf (logfp, "\nVendor Tag:\t%s\n", argv[1]);
    (void) fprintf (logfp, "Release Tags:\t");
    for (i = 2; i < argc; i++)
```

```
            (void) fprintf (logfp, "%s\n\t\t", argv[i]);
    (void) fprintf (logfp, "\n");

    /* Just Do It.  */
    err = import_descend (message, argv[1], argc - 2, argv + 2);
    if (conflicts)
    {
        if (!really_quiet)
        {
            char buf[80];
            sprintf (buf, "\n%d conflicts created by this import.\n",
                    conflicts);
            cvs_output (buf, 0);
            cvs_output ("Use the following command to help the merge:\n\n",
                    0);
            cvs_output ("\t", 1);
            cvs_output (program_name, 0);
            cvs_output (" checkout -j", 0);
            cvs_output (argv[1], 0);
            cvs_output (":yesterday -j", 0);
            cvs_output (argv[1], 0);
            cvs_output (" ", 1);
            cvs_output (argv[0], 0);
            cvs_output ("\n\n", 0);
        }

        (void) fprintf (logfp, "\n%d conflicts created by this import.\n",
                        conflicts);
        (void) fprintf (logfp,
                        "Use the following command to help the merge:\n\n");
        (void) fprintf (logfp, "\t%s checkout -j%s:yesterday -j%s %s\n\n",
                        program_name, argv[1], argv[1], argv[0]);
    }
    else
    {
        if (!really_quiet)
            cvs_output ("\nNo conflicts created by this import\n\n", 0);
        (void) fprintf (logfp, "\nNo conflicts created by this import\n\n");
    }

    /*
     * Write out the logfile and clean up.
     */
    ulist = getlist ();
    p = getnode ();
    p->type = UPDATE;
    p->delproc = update_delproc;
    p->key = xstrdup ("- Imported sources");
    li = (struct logfile_info *) xmalloc (sizeof (struct logfile_info));
    li->type = T_TITLE;
    li->tag = xstrdup (vbranch);
    li->rev_old = li->rev_new = NULL;
    p->data = (char *) li;
    (void) addnode (ulist, p);
    Update_Logfile (repository, message, logfp, ulist);
    dellist (&ulist);
    if (fclose (logfp) < 0)
        error (0, errno, "error closing %s", tmpfile);

    /* Make sure the temporary file goes away, even on systems that don't let
       you delete a file that's in use.  */
    if (CVS_UNLINK (tmpfile) < 0 && !existence_error (errno))
        error (0, errno, "cannot remove %s", tmpfile);
    free (tmpfile);

    if (message)
        free (message);
    free (repository);
    free (vbranch);
    free (vhead);

    return (err);
}

/*
 * process all the files in ".", then descend into other directories.
 */
static int
import_descend (message, vtag, targc, targv)
    char *message;
    char *vtag;
    int targc;
    char *targv[];
{
    DIR *dirp;
    struct dirent *dp;
    int err = 0;
    List *dirlist = NULL;

    /* first, load up any per-directory ignore lists */
```

```
    ign_add_file (CVSDOTIGNORE, 1);
    wrap_add_file (CVSDOTWRAPPER, 1);

    if ((dirp = CVS_OPENDIR (".")) == NULL)
    {
        err++;
    }
    else
    {
        while ((dp = readdir (dirp)) != NULL)
        {
            if (strcmp (dp->d_name, ".") == 0 || strcmp (dp->d_name, "..") == 0)
                continue;
#ifdef SERVER_SUPPORT
            /* CVS directories are created in the temp directory by
               server.c because it doesn't special-case import.  So
               don't print a message about them, regardless of -I!.  */
            if (server_active && strcmp (dp->d_name, CVSADM) == 0)
                continue;
#endif
            if (ign_name (dp->d_name))
            {
                add_log ('I', dp->d_name);
                continue;
            }

            if (
#ifdef DT_DIR
                (dp->d_type == DT_DIR
                 || (dp->d_type == DT_UNKNOWN && isdir (dp->d_name)))
#else
                isdir (dp->d_name)
#endif
                && !wrap_name_has (dp->d_name, WRAP_TOCVS)
                )
            {
                Node *n;

                if (dirlist == NULL)
                    dirlist = getlist();

                n = getnode();
                n->key = xstrdup (dp->d_name);
                addnode(dirlist, n);
            }
            else if (
#ifdef DT_DIR
                dp->d_type == DT_LNK || dp->d_type == DT_UNKNOWN &&
#endif
                islink (dp->d_name))
            {
                add_log ('L', dp->d_name);
                err++;
            }
            else
            {
#ifdef CLIENT_SUPPORT
                if (client_active)
                    err += client_process_import_file (message, dp->d_name,
                                                       vtag, targc, targv,
                                                       repository,
                                                       keyword_opt != NULL &&
                                                       keyword_opt[0] == 'b');
                else
#endif
                    err += process_import_file (message, dp->d_name,
                                                vtag, targc, targv);
            }
        }
        (void) closedir (dirp);
    }

    if (dirlist != NULL)
    {
        Node *head, *p;

        head = dirlist->list;
        for (p = head->next; p != head; p = p->next)
        {
            err += import_descend_dir (message, p->key, vtag, targc, targv);
        }

        dellist(&dirlist);
    }

    return (err);
}

/*
 * Process the argument import file.
```

```
 */
static int
process_import_file (message, vfile, vtag, targc, targv)
    char *message;
    char *vfile;
    char *vtag;
    int targc;
    char *targv[];
{
    char *rcs;
    int inattic = 0;

    rcs = xmalloc (strlen (repository) + strlen (vfile) + sizeof (RCSEXT)
                   + 5);
    (void) sprintf (rcs, "%s/%s%s", repository, vfile, RCSEXT);
    if (!isfile (rcs))
    {
        char *attic_name;

        attic_name = xmalloc (strlen (repository) + strlen (vfile) +
                              sizeof (CVSATTIC) + sizeof (RCSEXT) + 10);
        (void) sprintf (attic_name, "%s/%s/%s%s", repository, CVSATTIC,
                        vfile, RCSEXT);
        if (!isfile (attic_name))
        {
            int retval;
            char *free_opt = NULL;
            char *our_opt = keyword_opt;

            free (attic_name);
            /*
             * A new import source file; it doesn't exist as a ,v within the
             * repository nor in the Attic -- create it anew.
             */
            add_log ('N', vfile);

#ifdef SERVER_SUPPORT
            /* The most reliable information on whether the file is binary
               is what the client told us.  That is because if the client had
               the wrong idea about binaryness, it corrupted the file, so
               we might as well believe the client.  */
            if (server_active)
            {
                Node *node;
                List *entries;

                /* Reading all the entries for each file is fairly silly, and
                   probably slow.  But I am too lazy at the moment to do
                   anything else.  */
                entries = Entries_Open (0, NULL);
                node = findnode_fn (entries, vfile);
                if (node != NULL)
                {
                    Entnode *entdata = (Entnode *) node->data;
                    if (entdata->type == ENT_FILE)
                    {
                        assert (entdata->options[0] == '-'
                                && entdata->options[1] == 'k');
                        our_opt = xstrdup (entdata->options + 2);
                        free_opt = our_opt;
                    }
                }
                Entries_Close (entries);
            }
#endif

            retval = add_rcs_file (message, rcs, vfile, vhead, our_opt,
                                   vbranch, vtag, targc, targv,
                                   NULL, 0, NULL, logfp);
            if (free_opt != NULL)
                free (free_opt);
            free (rcs);
            return retval;
        }
        free (attic_name);
        inattic = 1;
    }

    free (rcs);
    /*
     * an rcs file exists. have to do things the official, slow, way.
     */
    return (update_rcs_file (message, vfile, vtag, targc, targv, inattic));
}

/*
 * The RCS file exists; update it by adding the new import file to the
 * (possibly already existing) vendor branch.
 */
static int
```

```
update_rcs_file (message, vfile, vtag, targc, targv, inattic)
    char *message;
    char *vfile;
    char *vtag;
    int targc;
    char *targv[];
    int inattic;
{
    Vers_TS *vers;
    int letter;
    char *tocvsPath;
    struct file_info finfo;

    memset (&finfo, 0, sizeof finfo);
    finfo.file = vfile;
    /* Not used, so don't worry about it.  */
    finfo.update_dir = NULL;
    finfo.fullname = finfo.file;
    finfo.repository = repository;
    finfo.entries = NULL;
    finfo.rcs = NULL;
    vers = Version_TS (&finfo, (char *) NULL, vbranch, (char *) NULL,
                       1, 0);
    if (vers->vn_rcs != NULL
        && !RCS_isdead(vers->srcfile, vers->vn_rcs))
    {
        int different;

        /*
         * The rcs file does have a revision on the vendor branch. Compare
         * this revision with the import file; if they match exactly, there
         * is no need to install the new import file as a new revision to the
         * branch.  Just tag the revision with the new import tags.
         *
         * This is to try to cut down the number of "C" conflict messages for
         * locally modified import source files.
         */
        tocvsPath = wrap_tocvs_process_file (vfile);
        /* FIXME: Why don't we pass tocvsPath to RCS_cmp_file if it is
           not NULL?  */
        different = RCS_cmp_file (vers->srcfile, vers->vn_rcs, "-ko", vfile);
        if (tocvsPath)
            if (unlink_file_dir (tocvsPath) < 0)
                error (0, errno, "cannot remove %s", tocvsPath);

        if (!different)
        {
            int retval = 0;

            /*
             * The two files are identical.  Just update the tags, print the
             * "U", signifying that the file has changed, but needs no
             * attention, and we're done.
             */
            if (add_tags (vers->srcfile, vfile, vtag, targc, targv))
                retval = 1;
            add_log ('U', vfile);
            freevers_ts (&vers);
            return (retval);
        }
    }

    /* We may have failed to parse the RCS file; check just in case */
    if (vers->srcfile == NULL ||
        add_rev (message, vers->srcfile, vfile, vers->vn_rcs) ||
        add_tags (vers->srcfile, vfile, vtag, targc, targv))
    {
        freevers_ts (&vers);
        return (1);
    }

    if (vers->srcfile->branch == NULL || inattic ||
        strcmp (vers->srcfile->branch, vbranch) != 0)
    {
        conflicts++;
        letter = 'C';
    }
    else
        letter = 'U';
    add_log (letter, vfile);

    freevers_ts (&vers);
    return (0);
}

/*
 * Add the revision to the vendor branch
 */
static int
add_rev (message, rcs, vfile, vers)
```

```
    char *message;
    RCSNode *rcs;
    char *vfile;
    char *vers;
{
    int locked, status, ierrno;
    char *tocvsPath;

    if (noexec)
        return (0);

    locked = 0;
    if (vers != NULL)
    {
        /* Before RCS_lock existed, we were directing stdout, as well as
           stderr, from the RCS command, to DEVNULL.  I wouldn't guess that
           was necessary, but I don't know for sure.  */
        /* Earlier versions of this function printed a 'fork failed' error
           when RCS_lock returned an error code.  That's not appropriate
           now that RCS_lock is librarified, but should the error text be
           preserved? */
        if (RCS_lock (rcs, vbranch, 1) != 0)
            return 1;
        locked = 1;
        RCS_rewrite (rcs, NULL, NULL);
    }
    tocvsPath = wrap_tocvs_process_file (vfile);

    status = RCS_checkin (rcs, tocvsPath == NULL ? vfile : tocvsPath,
                          message, vbranch,
                          (RCS_FLAGS_QUIET | RCS_FLAGS_KEEPFILE
                           | (use_file_modtime ? RCS_FLAGS_MODTIME : 0)));
    ierrno = errno;

    if ((tocvsPath != NULL) && (unlink_file_dir (tocvsPath) < 0))
        error (0, errno, "cannot remove %s", tocvsPath);

    if (status)
    {
        if (!noexec)
        {
            fperror (logfp, 0, status == -1 ? ierrno : 0,
                     "ERROR: Check-in of %s failed", rcs->path);
            error (0, status == -1 ? ierrno : 0,
                   "ERROR: Check-in of %s failed", rcs->path);
        }
        if (locked)
        {
            (void) RCS_unlock(rcs, vbranch, 0);
            RCS_rewrite (rcs, NULL, NULL);
        }
        return (1);
    }
    return (0);
}

/*
 * Add the vendor branch tag and all the specified import release tags to the
 * RCS file.  The vendor branch tag goes on the branch root (1.1.1) while the
 * vendor release tags go on the newly added leaf of the branch (1.1.1.1,
 * 1.1.1.2, ...).
 */
static int
add_tags (rcs, vfile, vtag, targc, targv)
    RCSNode *rcs;
    char *vfile;
    char *vtag;
    int targc;
    char *targv[];
{
    int i, ierrno;
    Vers_TS *vers;
    int retcode = 0;
    struct file_info finfo;

    if (noexec)
        return (0);

    if ((retcode = RCS_settag(rcs, vtag, vbranch)) != 0)
    {
        ierrno = errno;
        fperror (logfp, 0, retcode == -1 ? ierrno : 0,
                 "ERROR: Failed to set tag %s in %s", vtag, rcs->path);
        error (0, retcode == -1 ? ierrno : 0,
               "ERROR: Failed to set tag %s in %s", vtag, rcs->path);
        return (1);
    }
    RCS_rewrite (rcs, NULL, NULL);

    memset (&finfo, 0, sizeof finfo);
```

```
    finfo.file = vfile;
    /* Not used, so don't worry about it.  */
    finfo.update_dir = NULL;
    finfo.fullname = finfo.file;
    finfo.repository = repository;
    finfo.entries = NULL;
    finfo.rcs = NULL;
    vers = Version_TS (&finfo, NULL, vtag, NULL, 1, 0);
    for (i = 0; i < targc; i++)
    {
        if ((retcode = RCS_settag (rcs, targv[i], vers->vn_rcs)) == 0)
            RCS_rewrite (rcs, NULL, NULL);
        else
        {
            ierrno = errno;
            fperror (logfp, 0, retcode == -1 ? ierrno : 0,
                     "WARNING: Couldn't add tag %s to %s", targv[i],
                     rcs->path);
            error (0, retcode == -1 ? ierrno : 0,
                   "WARNING: Couldn't add tag %s to %s", targv[i],
                   rcs->path);
        }
    }
    freevers_ts (&vers);
    return (0);
}

/*
 * Stolen from rcs/src/rcsfnms.c, and adapted/extended.
 */
struct compair
{
    char *suffix, *comlead;
};

static const struct compair comtable[] =
{

/*
 * comtable pairs each filename suffix with a comment leader. The comment
 * leader is placed before each line generated by the $Log keyword. This
 * table is used to guess the proper comment leader from the working file's
 * suffix during initial ci (see InitAdmin()). Comment leaders are needed for
 * languages without multiline comments; for others they are optional.
 *
 * I believe that the comment leader is unused if you are using RCS 5.7, which
 * decides what leader to use based on the text surrounding the $Log keyword
 * rather than a specified comment leader.
 */
    {"a", "-- "},               /* Ada        */
    {"ada", "-- "},
    {"adb", "-- "},
    {"asm", ";; "},             /* assembler (MS-DOS) */
    {"ads", "-- "},             /* Ada        */
    {"bas", "' "},              /* Visual Basic code */
    {"bat", ":: "},             /* batch (MS-DOS) */
    {"body", "-- "},            /* Ada        */
    {"c", " * "},               /* C          */
    {"c++", "// "},             /* C++ in all its infinite guises */
    {"cc", "// "},
    {"cpp", "// "},
    {"cxx", "// "},
    {"m", "// "},               /* Objective-C */
    {"cl", ";;; "},             /* Common Lisp  */
    {"cmd", ":: "},             /* command (OS/2) */
    {"cmf", "c "},              /* CM Fortran  */
    {"cs", " * "},              /* C*         */
    {"csh", "# "},              /* shell      */
    {"dlg", " * "},             /* MS Windows dialog file */
    {"e", "# "},                /* efl        */
    {"epsf", "% "},             /* encapsulated postscript */
    {"epsi", "% "},             /* encapsulated postscript */
    {"el", "; "},               /* Emacs Lisp  */
    {"f", "c "},                /* Fortran    */
    {"for", "c "},
    {"frm", "' "},              /* Visual Basic form */
    {"h", " * "},               /* C-header  */
    {"hh", "// "},              /* C++ header  */
    {"hpp", "// "},
    {"hxx", "// "},
    {"in", "# "},               /* for Makefile.in */
    {"l", " * "},               /* lex (conflict between lex and
                                 * franzlisp) */
    {"mac", ";; "},             /* macro (DEC-10, MS-DOS, PDP-11,
                                 * VMS, etc) */
    {"mak", "# "},              /* makefile, e.g. Visual C++ */
    {"me", ".\\\" "},           /* me-macros t/nroff  */
    {"ml", "; "},               /* mocklisp  */
    {"mm", ".\\\" "},           /* mm-macros t/nroff  */
    {"ms", ".\\\" "},           /* ms-macros t/nroff  */
```

```
    {"man", ".\\\" "},          /* man-macros t/nroff  */
    {"1", ".\\\" "},            /* feeble attempt at man pages... */
    {"2", ".\\\" "},
    {"3", ".\\\" "},
    {"4", ".\\\" "},
    {"5", ".\\\" "},
    {"6", ".\\\" "},
    {"7", ".\\\" "},
    {"8", ".\\\" "},
    {"9", ".\\\" "},
    {"p", " * "},               /* pascal     */
    {"pas", " * "},
    {"pl", "# "},               /* perl (conflict with Prolog) */
    {"ps", "% "},               /* postscript  */
    {"psw", "% "},              /* postscript wrap */
    {"pswm", "% "},             /* postscript wrap */
    {"r", "# "},                /* ratfor      */
    {"rc", " * "},              /* Microsoft Windows resource file */
    {"red", "% "},              /* psl/rlisp   */
#ifdef sparc
    {"s", "! "},                /* assembler  */
#endif
#ifdef mc68000
    {"s", "| "},                /* assembler  */
#endif
#ifdef pdp11
    {"s", "/ "},                /* assembler  */
#endif
#ifdef vax
    {"s", "# "},                /* assembler  */
#endif
#ifdef __ksr__
    {"s", "# "},                /* assembler  */
    {"S", "# "},                /* Macro assembler */
#endif
    {"sh", "# "},               /* shell       */
    {"sl", "% "},               /* psl         */
    {"spec", "-- "},            /* Ada         */
    {"tex", "% "},              /* tex         */
    {"y", " * "},               /* yacc        */
    {"ye", " * "},              /* yacc-efl    */
    {"yr", " * "},              /* yacc-ratfor  */
    {"", "# "},                 /* default for empty suffix  */
    {NULL, "# "}                /* default for unknown suffix;  */
/* must always be last      */
};

static char *
get_comment (user)
    char *user;
{
    char *cp, *suffix;
    char *suffix_path;
    int i;
    char *retval;

    suffix_path = xmalloc (strlen (user) + 5);
    cp = strrchr (user, '.');
    if (cp != NULL)
    {
        cp++;

        /*
         * Convert to lower-case, since we are not concerned about the
         * case-ness of the suffix.
         */
        (void) strcpy (suffix_path, cp);
        for (cp = suffix_path; *cp; cp++)
            if (isupper (*cp))
                *cp = tolower (*cp);
        suffix = suffix_path;
    }
    else
        suffix = "";                /* will use the default */
    for (i = 0;; i++)
    {
        if (comtable[i].suffix == NULL)
        {
            /* Default.  Note we'll always hit this case before we
               ever return NULL.  */
            retval = comtable[i].comlead;
            break;
        }
        if (strcmp (suffix, comtable[i].suffix) == 0)
        {
            retval = comtable[i].comlead;
            break;
        }
    }
    free (suffix_path);
```

```
    return retval;
}

/* Create a new RCS file from scratch.

   This probably should be moved to rcs.c now that it is called from
   places outside import.c.

   Return value is 0 for success, or nonzero for failure (in which
   case an error message will have already been printed).  */
int
add_rcs_file (message, rcs, user, add_vhead, key_opt,
              add_vbranch, vtag, targc, targv,
              desctext, desclen, remote_bp, add_logfp)
    /* Log message for the addition.  Not used if add_vhead == NULL.  */
    char *message;
    /* Filename of the RCS file to create.  */
    char *rcs;
    /* Filename of the file to serve as the contents of the initial
       revision.  Even if add_vhead is NULL, we use this to determine
       the modes to give the new RCS file.  */
    char *user;

    /* Revision number of head that we are adding.  Normally 1.1 but
       could be another revision as long as ADD_VBRANCH is a branch
       from it.  If NULL, then just add an empty file without any
       revisions (similar to the one created by "rcs -i").  */
    char *add_vhead;

    /* Keyword expansion mode, e.g., "b" for binary.  NULL means the
       default behavior.  */
    char *key_opt;

    /* Vendor branch to import to, or NULL if none.  If non-NULL, then
       vtag should also be non-NULL.  */
    char *add_vbranch;
    char *vtag;
    int targc;
    char *targv[];

    /* If non-NULL, description for the file.  If NULL, the description
       will be empty.  */
    char *desctext;
    size_t desclen;

    /* Remote branchpoint, NULL if none */
    char* remote_bp;

    /* Write errors to here as well as via error (), or NULL if we should
       use only error ().  */
    FILE *add_logfp;
{
    FILE *fprcs, *fpuser;
    struct stat sb;
    struct tm *ftm;
    time_t now;
    char altdate1[MAXDATELEN];
    char *author;
    int i, ierrno, err = 0;
    mode_t mode;
    char *tocvsPath;
    char *userfile;
    char *local_opt = key_opt;
    char *free_opt = NULL;
    mode_t file_type;

    if (noexec)
        return (0);

    /* Note that as the code stands now, the -k option overrides any
       settings in wrappers (whether CVSROOT/cvswrappers, -W, or
       whatever).  Some have suggested this should be the other way
       around.  As far as I know the documentation doesn't say one way
       or the other.  Before making a change of this sort, should think
       about what is best, document it (in cvs.texinfo and NEWS), &c.  */

    if (local_opt == NULL)
    {
        if (wrap_name_has (user, WRAP_RCSOPTION))
        {
            local_opt = free_opt = wrap_rcsoption (user, 0);
        }
    }

    tocvsPath = wrap_tocvs_process_file (user);
    userfile = (tocvsPath == NULL ? user : tocvsPath);

    /* Opening in text mode is probably never the right thing for the
       server (because the protocol encodes text files in a fashion
       which does not depend on what the client or server OS is, as
```

```
    documented in cvsclient.texi), but as long as the server just
    runs on unix it is a moot point.  */

/* If PreservePermissions is set, then make sure that the file
   is a plain file before trying to open it.  Longstanding (although
   often unpopular) CVS behavior has been to follow symlinks, so we
   maintain that behavior if PreservePermissions is not on.

   NOTE: this error message used to be 'cannot fstat', but is now
   'cannot lstat'.  I don't see a way around this, since we must
   stat the file before opening it. -twp */

if (CVS_LSTAT (userfile, &sb) < 0)
    error (1, errno, "cannot lstat %s", user);
file_type = sb.st_mode & S_IFMT;

fpuser = NULL;
if (!preserve_perms || file_type == S_IFREG)
{
    fpuser = CVS_FOPEN (userfile,
                        ((local_opt != NULL && strcmp (local_opt, "b") == 0)
                         ? "rb"
                         : "r")
        );
    if (fpuser == NULL)
    {
        /* not fatal, continue import */
        if (add_logfp != NULL)
            fperror (add_logfp, 0, errno,
                     "ERROR: cannot read file %s", userfile);
        error (0, errno, "ERROR: cannot read file %s", userfile);
        goto read_error;
    }
}

fprcs = CVS_FOPEN (rcs, "w+b");
if (fprcs == NULL)
{
    ierrno = errno;
    goto write_error_noclose;
}

/*
 * putadmin()
 */
if (add_vhead != NULL)
{
    if (fprintf (fprcs, "head     %s;\012", add_vhead) < 0)
        goto write_error;
}
else
{
    if (fprintf (fprcs, "head     ;\012") < 0)
        goto write_error;
}

if (remote_bp != NULL) {
  if (fprintf (fprcs, "remote_branchpoint     %s;\012", remote_bp) < 0)
    goto write_error;
} else {
  if (fprintf (fprcs, "remote_branchpoint     ;\012") < 0)
    goto write_error;
}

if (add_vbranch != NULL)
{
    if (fprintf (fprcs, "branch   %s;\012", add_vbranch) < 0)
        goto write_error;
}
if (fprintf (fprcs, "access   ;\012") < 0 ||
    fprintf (fprcs, "symbols  ") < 0)
{
    goto write_error;
}

for (i = targc - 1; i >= 0; i--)
{
    /* RCS writes the symbols backwards */
    assert (add_vbranch != NULL);
    if (fprintf (fprcs, "%s:%s.1 ", targv[i], add_vbranch) < 0)
        goto write_error;
}

if (add_vbranch != NULL)
{
    if (fprintf (fprcs, "%s:%s", vtag, add_vbranch) < 0)
        goto write_error;
}
if (fprintf (fprcs, ";\012") < 0)
    goto write_error;
```

```
1080    if (fprintf (fprcs, "locks    ; strict;\012") < 0 ||
            /* XXX - make sure @ processing works in the RCS file */
            fprintf (fprcs, "comment  @%s@;\012", get_comment (user)) < 0)
        {
            goto write_error;
        }

        if (local_opt != NULL)
        {
            if (fprintf (fprcs, "expand   @%s@;\012", local_opt) < 0)
1090        {
                goto write_error;
            }
        }

        if (fprintf (fprcs, "\012") < 0)
          goto write_error;

        /* Write the revision(s), with the date and author and so on
           (that is "delta" rather than "deltatext" from rcsfile(5)).  */
1100    if (add_vhead != NULL)
        {
            if (use_file_modtime)
                now = sb.st_mtime;
            else
                (void) time (&now);
            ftm = gmtime (&now);
            (void) sprintf (altdate1, DATEFORM,
                            ftm->tm_year + (ftm->tm_year < 100 ? 0 : 1900),
                            ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
1110                        ftm->tm_min, ftm->tm_sec);
            author = getcaller ();

            if (fprintf (fprcs, "\012%s\012", add_vhead) < 0 ||
            fprintf (fprcs, "date     %s;  author %s;  state Exp;\012",
                    altdate1, author) < 0)
            goto write_error;

            if (fprintf (fprcs, "branches") < 0)
                goto write_error;
1120        if (add_vbranch != NULL)
            {
                if (fprintf (fprcs, " %s.1", add_vbranch) < 0)
                    goto write_error;
            }
            if (fprintf (fprcs, ";\012") < 0)
                goto write_error;


            if (fprintf (fprcs, "remote-branches") < 0)
1130            goto write_error;
            if (fprintf (fprcs, ";\012") < 0)
                goto write_error;

            if (fprintf (fprcs, "next     ;\012") < 0)
                goto write_error;

    #ifdef PRESERVE_PERMISSIONS_SUPPORT
            /* Store initial permissions if necessary. */
            if (preserve_perms)
1140        {
                if (file_type == S_IFLNK)
                {
                    char *link = xreadlink (userfile);
                    if (fprintf (fprcs, "symlink\t@") < 0 ||
                        expand_at_signs (link, strlen (link), fprcs) < 0 ||
                        fprintf (fprcs, "@;\012") < 0)
                        goto write_error;
                    free (link);
                }
1150            else
                {
                    if (fprintf (fprcs, "owner\t%u;\012", sb.st_uid) < 0)
                        goto write_error;
                    if (fprintf (fprcs, "group\t%u;\012", sb.st_gid) < 0)
                        goto write_error;
                    if (fprintf (fprcs, "permissions\t%o;\012",
                                 sb.st_mode & 07777) < 0)
                        goto write_error;
                    switch (file_type)
1160                {
                        case S_IFREG: break;
                        case S_IFCHR:
                        case S_IFBLK:
                            if (fprintf (fprcs, "special\t%s %lu;\012",
                                         (file_type == S_IFCHR
                                          ? "character"
                                          : "block"),
                                         (unsigned long) sb.st_rdev) < 0)
```

```
                    goto write_error;
                break;
            default:
                error (0, 0,
                       "can't import %s: unknown kind of special file",
                       userfile);
            }
        }
    }
#endif

    if (add_vbranch != NULL)
    {
        if (fprintf (fprcs, "\012%s.1\012", add_vbranch) < 0 ||
            fprintf (fprcs, "date     %s;  author %s;  state Exp;\012",
                     altdate1, author) < 0 ||
            fprintf (fprcs, "branches ;\012") < 0 ||
            fprintf (fprcs, "next     ;\012") < 0)
            goto write_error;

#ifdef PRESERVE_PERMISSIONS_SUPPORT
        /* Store initial permissions if necessary. */
        if (preserve_perms)
        {
            if (file_type == S_IFLNK)
            {
                char *link = xreadlink (userfile);
                if (fprintf (fprcs, "symlink\t@") < 0 ||
                    expand_at_signs (link, strlen (link), fprcs) < 0 ||
                    fprintf (fprcs, "@;\012") < 0)
                    goto write_error;
                free (link);
            }
            else
            {
                if (fprintf (fprcs, "owner\t%u;\012", sb.st_uid) < 0 ||
                    fprintf (fprcs, "group\t%u;\012", sb.st_gid) < 0 ||
                    fprintf (fprcs, "permissions\t%o;\012",
                             sb.st_mode & 07777) < 0)
                    goto write_error;

                switch (file_type)
                {
                    case S_IFREG: break;
                    case S_IFCHR:
                    case S_IFBLK:
                        if (fprintf (fprcs, "special\t%s %lu;\012",
                                     (file_type == S_IFCHR
                                      ? "character"
                                      : "block"),
                                     (unsigned long) sb.st_rdev) < 0)
                            goto write_error;
                        break;
                    default:
                        error (0, 0,
                          "cannot import %s: special file of unknown type",
                          userfile);
                }
            }
        }
#endif

        if (fprintf (fprcs, "\012") < 0)
            goto write_error;
    }

    /* Now write the description (possibly empty).  */
    if (fprintf (fprcs, "\012desc\012") < 0 ||
        fprintf (fprcs, "@") < 0)
        goto write_error;
    if (desctext != NULL)
    {
        /* The use of off_t not size_t for the second argument is very
           strange, since we are dealing with something which definitely
           fits in memory.  */
        if (expand_at_signs (desctext, (off_t) desclen, fprcs) < 0)
            goto write_error;
    }
    if (fprintf (fprcs, "@\012\012\012") < 0)
        goto write_error;

    /* Now write the log messages and contents for the revision(s) (that
       is, "deltatext" rather than "delta" from rcsfile(5)).  */
    if (add_vhead != NULL)
    {
        if (fprintf (fprcs, "\012%s\012", add_vhead) < 0 ||
            fprintf (fprcs, "log\012@") < 0)
            goto write_error;
        if (add_vbranch != NULL)
```

```
    {
1260    /* We are going to put the log message in the revision on the
       branch.  So putting it here too seems kind of redundant, I
       guess (and that is what CVS has always done, anyway).  */
    if (fprintf (fprcs, "Initial revision\012") < 0)
        goto write_error;
    }
    else
    {
    if (expand_at_signs (message, (off_t) strlen (message), fprcs) < 0)
        goto write_error;
1270    }
    if (fprintf (fprcs, "@\012") < 0 ||
        fprintf (fprcs, "text\012@") < 0)
    {
        goto write_error;
    }

    /* Now copy over the contents of the file, expanding at signs.
       If preserve_perms is set, do this only for regular files. */
    if (!preserve_perms || file_type == S_IFREG)
1280    {
        char buf[8192];
        unsigned int len;

        while (1)
        {
            len = fread (buf, 1, sizeof buf, fpuser);
            if (len == 0)
            {
                if (ferror (fpuser))
1290                error (1, errno, "cannot read file %s for copying",
                           user);
                break;
            }
            if (expand_at_signs (buf, len, fprcs) < 0)
                goto write_error;
        }
    }
    if (fprintf (fprcs, "@\012\012") < 0)
        goto write_error;
1300    if (add_vbranch != NULL)
    {
        if (fprintf (fprcs, "\012%s.1\012", add_vbranch) < 0 ||
            fprintf (fprcs, "log\012@") < 0 ||
            expand_at_signs (message,
                             (off_t) strlen (message), fprcs) < 0 ||
            fprintf (fprcs, "@\012text\012") < 0 ||
            fprintf (fprcs, "@@\012") < 0)
            goto write_error;
    }
1310}

if (fclose (fprcs) == EOF)
{
    ierrno = errno;
    goto write_error_noclose;
}
/* Close fpuser only if we opened it to begin with. */
if (fpuser != NULL)
{
1320    if (fclose (fpuser) < 0)
        error (0, errno, "cannot close %s", user);
}

/*
 * Fix the modes on the RCS files.  The user modes of the original
 * user file are propagated to the group and other modes as allowed
 * by the repository umask, except that all write permissions are
 * turned off.
 */
1330mode = (sb.st_mode |
       (sb.st_mode & S_IRWXU) >> 3 |
       (sb.st_mode & S_IRWXU) >> 6) &
      ~cvsumask &
      ~(S_IWRITE | S_IWGRP | S_IWOTH);
if (chmod (rcs, mode) < 0)
{
    ierrno = errno;
    if (add_logfp != NULL)
        fperror (add_logfp, 0, ierrno,
1340             "WARNING: cannot change mode of file %s", rcs);
    error (0, ierrno, "WARNING: cannot change mode of file %s", rcs);
    err++;
}
if (tocvsPath)
    if (unlink_file_dir (tocvsPath) < 0)
        error (0, errno, "cannot remove %s", tocvsPath);
if (free_opt != NULL)
    free (free_opt);
```

```
    return (err);

write_error:
    ierrno = errno;
    if (fclose (fprcs) < 0)
        error (0, errno, "cannot close %s", rcs);
write_error_noclose:
    if (fclose (fpuser) < 0)
        error (0, errno, "cannot close %s", user);
    if (add_logfp != NULL)
        fperror (add_logfp, 0, ierrno, "ERROR: cannot write file %s", rcs);
    error (0, ierrno, "ERROR: cannot write file %s", rcs);
    if (ierrno == ENOSPC)
    {
        if (CVS_UNLINK (rcs) < 0)
            error (0, errno, "cannot remove %s", rcs);
        if (add_logfp != NULL)
            fperror (add_logfp, 0, 0, "ERROR: out of space - aborting");
        error (1, 0, "ERROR: out of space - aborting");
    }
read_error:
    if (tocvsPath)
        if (unlink_file_dir (tocvsPath) < 0)
            error (0, errno, "cannot remove %s", tocvsPath);

    if (free_opt != NULL)
        free (free_opt);

    return (err + 1);
}

/*
 * Write SIZE bytes at BUF to FP, expanding  signs into double
 * signs.  If an error occurs, return a negative value and set errno
 * to indicate the error.  If not, return a nonnegative value.
 */
int
expand_at_signs (buf, size, fp)
    char *buf;
    off_t size;
    FILE *fp;
{
    register char *cp, *next;

    cp = buf;
    while ((next = memchr (cp, '@', size)) != NULL)
    {
        int len;

        ++next;
        len = next - cp;
        if (fwrite (cp, 1, len, fp) != len)
            return EOF;
        if (putc ('@', fp) == EOF)
            return EOF;
        cp = next;
        size -= len;
    }

    if (fwrite (cp, 1, size, fp) != size)
        return EOF;

    return 1;
}

/*
 * Write an update message to (potentially) the screen and the log file.
 */
static void
add_log (ch, fname)
    int ch;
    char *fname;
{
    if (!really_quiet)                /* write to terminal */
    {
        char buf[2];
        buf[0] = ch;
        buf[1] = ' ';
        cvs_output (buf, 2);
        if (repos_len)
        {
            cvs_output (repository + repos_len + 1, 0);
            cvs_output ("/", 1);
        }
        else if (repository[0] != '\0')
        {
            cvs_output (repository, 0);
            cvs_output ("/", 1);
        }
        cvs_output (fname, 0);
```

```
        cvs_output ("\n", 1);
    }

    if (repos_len)                  /* write to logfile */
        (void) fprintf (logfp, "%c %s/%s\n", ch,
                        repository + repos_len + 1, fname);
    else if (repository[0])
        (void) fprintf (logfp, "%c %s/%s\n", ch, repository, fname);
    else
        (void) fprintf (logfp, "%c %s\n", ch, fname);
}

/*
 * This is the recursive function that walks the argument directory looking
 * for sub-directories that have CVS administration files in them and updates
 * them recursively.
 *
 * Note that we do not follow symbolic links here, which is a feature!
 */
static int
import_descend_dir (message, dir, vtag, targc, targv)
    char *message;
    char *dir;
    char *vtag;
    int targc;
    char *targv[];
{
    struct saved_cwd cwd;
    char *cp;
    int ierrno, err;
    char *rcs = NULL;

    if (islink (dir))
        return (0);
    if (save_cwd (&cwd))
    {
        fperror (logfp, 0, 0, "ERROR: cannot get working directory");
        return (1);
    }

    /* Concatenate DIR to the end of REPOSITORY.  */
    if (repository[0] == '\0')
    {
        char *new = xstrdup (dir);
        free (repository);
        repository = new;
    }
    else
    {
        char *new = xmalloc (strlen (repository) + strlen (dir) + 10);
        strcpy (new, repository);
        (void) strcat (new, "/");
        (void) strcat (new, dir);
        free (repository);
        repository = new;
    }

#ifdef CLIENT_SUPPORT
    if (!quiet && !client_active)
#else
    if (!quiet)
#endif
        error (0, 0, "Importing %s", repository);

    if ( CVS_CHDIR (dir) < 0)
    {
        ierrno = errno;
        fperror (logfp, 0, ierrno, "ERROR: cannot chdir to %s", repository);
        error (0, ierrno, "ERROR: cannot chdir to %s", repository);
        err = 1;
        goto out;
    }
#ifdef CLIENT_SUPPORT
    if (!client_active && !isdir (repository))
#else
    if (!isdir (repository))
#endif
    {
        rcs = xmalloc (strlen (repository) + sizeof (RCSEXT) + 5);
        (void) sprintf (rcs, "%s%s", repository, RCSEXT);
        if (isfile (repository) || isfile(rcs))
        {
            fperror (logfp, 0, 0,
                     "ERROR: %s is a file, should be a directory!",
                     repository);
            error (0, 0, "ERROR: %s is a file, should be a directory!",
                   repository);
            err = 1;
            goto out;
        }
```

```
        if (noexec == 0 && CVS_MKDIR (repository, 0777) < 0)
        {
            ierrno = errno;
            fperror (logfp, 0, ierrno,
                     "ERROR: cannot mkdir %s -- not added", repository);
            error (0, ierrno,
                   "ERROR: cannot mkdir %s -- not added", repository);
            err = 1;
            goto out;
        }
    }
    err = import_descend (message, vtag, targc, targv);
 out:
    if (rcs != NULL)
        free (rcs);
    if ((cp = strrchr (repository, '/')) != NULL)
        *cp = '\0';
    else
        repository[0] = '\0';
    if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);
    return (err);
}
```

A.32 lock.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Set Lock
 *
 * Lock file support for CVS.
 */

/* The node Concurrency in doc/cvs.texinfo has a brief introduction to
   how CVS locks function, and some of the user-visible consequences of
   their existence.  Here is a summary of why they exist (and therefore,
   the consequences of hacking CVS to read a repository without creating
   locks):

   There are two uses.  One is the ability to prevent there from being
   two writers at the same time.  This is necessary for any number of
   reasons (fileattr code, probably others).  Commit needs to lock the
   whole tree so that nothing happens between the up-to-date check and
   the actual checkin.

   The second use is the ability to ensure that there is not a writer
   and a reader at the same time (several readers are allowed).  Reasons
   for this are:

   * Readlocks ensure that once CVS has found a collection of rcs
   files using Find_Names, the files will still exist when it reads
   them (they may have moved in or out of the attic).

   * Readlocks provide some modicum of consistency, although this is
   kind of limited--see the node Concurrency in cvs.texinfo.

   * Readlocks ensure that the RCS file does not change between
   RCS_parse and RCS_reparsercsfile time.  This one strikes me as
   important, although I haven't thought up what bad scenarios might
   be.

   * Readlocks ensure that we won't find the file in the state in
   which it is in between the calls to add_rcs_file and RCS_checkin in
   commit.c (when a file is being added).  This state is a state in
   which the RCS file parsing routines in rcs.c cannot parse the file.

   * Readlocks ensure that a reader won't try to look at a
   half-written fileattr file (fileattr is not updated atomically).

   (see also the description of anonymous read-only access in
   "Password authentication security" node in doc/cvs.texinfo).

   While I'm here, I'll try to summarize a few random suggestions
   which periodically get made about how locks might be different:

   1.  Check for EROFS.  Maybe useful, although in the presence of NFS
   EROFS does *not* mean that the file system is unchanging.

   2.  Provide a means to put the cvs locks in some directory apart from
   the repository (CVSROOT/locks; a -l option in modules; etc.).

   3.  Provide an option to disable locks for operations which only
   read (see above for some of the consequences).

   4.  Have a server internally do the locking.  Probably a good
   long-term solution, and many people have been working hard on code
   changes which would eventually make it possible to have a server
   which can handle various connections in one process, but there is
   much, much work still to be done before this is feasible.

   5.  Like #4 but use shared memory or something so that the servers
   merely need to all be on the same machine.  This is a much smaller
   change to CVS (it functions much like #2; shared memory might be an
   unneeded complication although it presumably would be faster).  */

#include "cvs.h"

struct lock {
    /* This is the directory in which we may have a lock named by the
       readlock variable, a lock named by the writelock variable, and/or
       a lock named CVSLCK.  The storage is not allocated along with the
       struct lock; it is allocated by the Reader_Lock caller or in the
       case of writelocks, it is just a pointer to the storage allocated
       for the ->key field.  */
    char *repository;
    /* Do we have a lock named CVSLCK?  */
    int have_lckdir;
    /* Note there is no way of knowing whether the readlock and writelock
       exist.  The code which sets the locks doesn't use SIG_beginCrSect
```

```
        to set a flag like we do for CVSLCK.  */
};

static void remove_locks PROTO((void));
static int readers_exist PROTO((char *repository));
static int set_lock PROTO ((struct lock *lock, int will_wait));
static void clear_lock PROTO ((struct lock *lock));
static void set_lockers_name PROTO((struct stat *statp));
static int set_writelock_proc PROTO((Node * p, void *closure));
static int unlock_proc PROTO((Node * p, void *closure));
static int write_lock PROTO ((struct lock *lock));
static void lock_simple_remove PROTO ((struct lock *lock));
static void lock_wait PROTO((char *repository));
static void lock_obtained PROTO((char *repository));

/* Malloc'd array containing the username of the whoever has the lock.
   Will always be non-NULL in the cases where it is needed.  */
static char *lockers_name;
/* Malloc'd array specifying name of a readlock within a directory.
   Or NULL if none.  */
static char *readlock;
/* Malloc'd array specifying name of a writelock within a directory.
   Or NULL if none.  */
static char *writelock;
/* Malloc'd array specifying the name of a CVSLCK file (absolute pathname).
   Will always be non-NULL in the cases where it is used.  */
static char *masterlock;
static List *locklist;

#define L_OK      0             /* success */
#define L_ERROR 1               /* error condition */
#define L_LOCKED 2              /* lock owned by someone else */

/* This is the (single) readlock which is set by Reader_Lock.  The
   repository field is NULL if there is no such lock.  */
static struct lock global_readlock;

/* List of locks set by lock_tree_for_write.  This is redundant
   with locklist, sort of.  */
static List *lock_tree_list;

/* If we set locks with lock_dir_for_write, then locked_dir contains
   the malloc'd name of the repository directory which we have locked.
   locked_list is the same thing packaged into a list and is redundant
   with locklist the same way that lock_tree_list is.  */
static char *locked_dir;
static List *locked_list;

/*
 * Clean up all outstanding locks
 */
void
Lock_Cleanup ()
{
    remove_locks ();

    dellist (&lock_tree_list);

    if (locked_dir != NULL)
    {
        dellist (&locked_list);
        free (locked_dir);
        locked_dir = NULL;
        locked_list = NULL;
    }
}

/*
 * Remove locks without discarding the lock information
 */
static void
remove_locks ()
{
    /* clean up simple locks (if any) */
    if (global_readlock.repository != NULL)
    {
        lock_simple_remove (&global_readlock);
        global_readlock.repository = NULL;
    }

    /* clean up multiple locks (if any) */
    if (locklist != (List *) NULL)
    {
        (void) walklist (locklist, unlock_proc, NULL);
        locklist = (List *) NULL;
    }
}

/*
 * walklist proc for removing a list of locks
```

```
 */
180 static int
unlock_proc (p, closure)
    Node *p;
    void *closure;
{
    lock_simple_remove ((struct lock *)p->data);
    return (0);
}

/* Remove the lock files.  */
190 static void
lock_simple_remove (lock)
    struct lock *lock;
{
    char *tmp;

    /* If readlock is set, the lock directory *might* have been created, but
       since Reader_Lock doesn't use SIG_beginCrSect the way that set_lock
       does, we don't know that.  That is why we need to check for
       existence_error here.  */
200 if (readlock != NULL)
    {
        tmp = xmalloc (strlen (lock->repository) + strlen (readlock) + 10);
        (void) sprintf (tmp, "%s/%s", lock->repository, readlock);
        if ( CVS_UNLINK (tmp) < 0 && ! existence_error (errno))
            error (0, errno, "failed to remove lock %s", tmp);
        free (tmp);
    }

    /* If writelock is set, the lock directory *might* have been created, but
210    since write_lock doesn't use SIG_beginCrSect the way that set_lock
       does, we don't know that.  That is why we need to check for
       existence_error here.  */
    if (writelock != NULL)
    {
        tmp = xmalloc (strlen (lock->repository) + strlen (writelock) + 10);
        (void) sprintf (tmp, "%s/%s", lock->repository, writelock);
        if ( CVS_UNLINK (tmp) < 0 && ! existence_error (errno))
            error (0, errno, "failed to remove lock %s", tmp);
        free (tmp);
220 }

    if (lock->have_lckdir)
    {
        tmp = xmalloc (strlen (lock->repository) + sizeof (CVSLCK) + 10);
        (void) sprintf (tmp, "%s/%s", lock->repository, CVSLCK);
        SIG_beginCrSect ();
        if (CVS_RMDIR (tmp) < 0)
            error (0, errno, "failed to remove lock dir %s", tmp);
        lock->have_lckdir = 0;
230     SIG_endCrSect ();
        free (tmp);
    }
}

/*
 * Create a lock file for readers
 */
int
Reader_Lock (xrepository)
240 char *xrepository;
{
    int err = 0;
    FILE *fp;
    char *tmp;

    if (noexec)
        return (0);

    /* we only do one directory at a time for read locks! */
250 if (global_readlock.repository != NULL)
    {
        error (0, 0, "Reader_Lock called while read locks set - Help!");
        return (1);
    }

    if (readlock == NULL)
    {
        readlock = xmalloc (strlen (hostname) + sizeof (CVSRFL) + 40);
        (void) sprintf (readlock,
260 #ifdef HAVE_LONG_FILE_NAMES
                        "%s.%s.%ld", CVSRFL, hostname,
#else
                        "%s.%ld", CVSRFL,
#endif
                        (long) getpid ());
    }

    /* remember what we're locking (for Lock_Cleanup) */
```

```
    global_readlock.repository = xrepository;

    /* get the lock dir for our own */
    if (set_lock (&global_readlock, 1) != L_OK)
    {
        error (0, 0, "failed to obtain dir lock in repository `%s'",
               xrepository);
        if (readlock != NULL)
            free (readlock);
        readlock = NULL;
        /* We don't set global_readlock.repository to NULL.  I think this
           only works because recurse.c will give a fatal error if we return
           a nonzero value.  */
        return (1);
    }

    /* write a read-lock */
    tmp = xmalloc (strlen (xrepository) + strlen (readlock) + 10);
    (void) sprintf (tmp, "%s/%s", xrepository, readlock);
    if ((fp = CVS_FOPEN (tmp, "w+")) == NULL || fclose (fp) == EOF)
    {
        error (0, errno, "cannot create read lock in repository `%s'",
               xrepository);
        if (readlock != NULL)
            free (readlock);
        readlock = NULL;
        err = 1;
    }
    free (tmp);

    /* free the lock dir */
    clear_lock (&global_readlock);

    return (err);
}

/*
 * Lock a list of directories for writing
 */
static char *lock_error_repos;
static int lock_error;

static int Writer_Lock PROTO ((List * list));

static int
Writer_Lock (list)
    List *list;
{
    char *wait_repos;

    if (noexec)
        return (0);

    /* We only know how to do one list at a time */
    if (locklist != (List *) NULL)
    {
        error (0, 0, "Writer_Lock called while write locks set - Help!");
        return (1);
    }

    wait_repos = NULL;
    for (;;)
    {
        /* try to lock everything on the list */
        lock_error = L_OK;      /* init for set_writelock_proc */
        lock_error_repos = (char *) NULL; /* init for set_writelock_proc */
        locklist = list;        /* init for Lock_Cleanup */
        if (lockers_name != NULL)
            free (lockers_name);
        lockers_name = xstrdup ("unknown");

        (void) walklist (list, set_writelock_proc, NULL);

        switch (lock_error)
        {
            case L_ERROR:       /* Real Error */
                if (wait_repos != NULL)
                    free (wait_repos);
                Lock_Cleanup ();  /* clean up any locks we set */
                error (0, 0, "lock failed - giving up");
                return (1);

            case L_LOCKED:      /* Someone already had a lock */
                remove_locks ();    /* clean up any locks we set */
                lock_wait (lock_error_repos); /* sleep a while and try again */
                wait_repos = xstrdup (lock_error_repos);
                continue;

            case L_OK:          /* we got the locks set */
                if (wait_repos != NULL)
```

```
            {
                lock_obtained (wait_repos);
                free (wait_repos);
            }
            return (0);

        default:
            if (wait_repos != NULL)
                free (wait_repos);
            error (0, 0, "unknown lock status %d in Writer_Lock",
                   lock_error);
            return (1);
        }
    }
}

/*
 * walklist proc for setting write locks
 */
static int
set_writelock_proc (p, closure)
    Node *p;
    void *closure;
{
    /* if some lock was not OK, just skip this one */
    if (lock_error != L_OK)
        return (0);

    /* apply the write lock */
    lock_error_repos = p->key;
    lock_error = write_lock ((struct lock *)p->data);
    return (0);
}

/*
 * Create a lock file for writers returns L_OK if lock set ok, L_LOCKED if
 * lock held by someone else or L_ERROR if an error occurred
 */
static int
write_lock (lock)
    struct lock *lock;
{
    int status;
    FILE *fp;
    char *tmp;

    if (writelock == NULL)
    {
        writelock = xmalloc (strlen (hostname) + sizeof (CVSWFL) + 40);
        (void) sprintf (writelock,
#ifdef HAVE_LONG_FILE_NAMES
                        "%s.%s.%ld", CVSWFL, hostname,
#else
                        "%s.%ld", CVSWFL,
#endif
                        (long) getpid());
    }

    /* make sure the lock dir is ours (not necessarily unique to us!) */
    status = set_lock (lock, 0);
    if (status == L_OK)
    {
        /* we now own a writer - make sure there are no readers */
        if (readers_exist (lock->repository))
        {
            /* clean up the lock dir if we created it */
            if (status == L_OK)
            {
                clear_lock (lock);
            }

            /* indicate we failed due to read locks instead of error */
            return (L_LOCKED);
        }

        /* write the write-lock file */
        tmp = xmalloc (strlen (lock->repository) + strlen (writelock) + 10);
        (void) sprintf (tmp, "%s/%s", lock->repository, writelock);
        if ((fp = CVS_FOPEN (tmp, "w+")) == NULL || fclose (fp) == EOF)
        {
            int xerrno = errno;

            if ( CVS_UNLINK (tmp) < 0 && ! existence_error (errno))
                error (0, errno, "failed to remove lock %s", tmp);

            /* free the lock dir if we created it */
            if (status == L_OK)
            {
                clear_lock (lock);
            }
```

```
            /* return the error */
            error (0, xerrno, "cannot create write lock in repository `%s'",
                   lock->repository);
            free (tmp);
            return (L_ERROR);
        }
        free (tmp);
        return (L_OK);
    }
    else
        return (status);
}

/*
 * readers_exist() returns 0 if there are no reader lock files remaining in
 * the repository; else 1 is returned, to indicate that the caller should
 * sleep a while and try again.
 */
static int
readers_exist (repository)
    char *repository;
{
    char *line;
    DIR *dirp;
    struct dirent *dp;
    struct stat sb;
    int ret = 0;

#ifdef CVS_FUDGELOCKS
again:
#endif

    if ((dirp = CVS_OPENDIR (repository)) == NULL)
        error (1, 0, "cannot open directory %s", repository);

    errno = 0;
    while ((dp = readdir (dirp)) != NULL)
    {
        if (CVS_FNMATCH (CVSRFLPAT, dp->d_name, 0) == 0)
        {
#ifdef CVS_FUDGELOCKS
            time_t now;
            (void) time (&now);
#endif

            line = xmalloc (strlen (repository) + strlen (dp->d_name) + 5);
            (void) sprintf (line, "%s/%s", repository, dp->d_name);
            if ( CVS_STAT (line, &sb) != -1)
            {
#ifdef CVS_FUDGELOCKS
                /*
                 * If the create time of the file is more than CVSLCKAGE
                 * seconds ago, try to clean-up the lock file, and if
                 * successful, re-open the directory and try again.
                 */
                if (now >= (sb.st_ctime + CVSLCKAGE) && CVS_UNLINK (line) != -1)
                {
                    (void) closedir (dirp);
                    free (line);
                    goto again;
                }
#endif
                set_lockers_name (&sb);
            }
            else
            {
                /* If the file doesn't exist, it just means that it disappeared
                   between the time we did the readdir and the time we did
                   the stat.  */
                if (!existence_error (errno))
                    error (0, errno, "cannot stat %s", line);
            }
            errno = 0;
            free (line);

            ret = 1;
            break;
        }
        errno = 0;
    }
    if (errno != 0)
        error (0, errno, "error reading directory %s", repository);

    closedir (dirp);
    return (ret);
}

/*
 * Set the static variable lockers_name appropriately, based on the stat
```

```
 * structure passed in.
 */
static void
set_lockers_name (statp)
    struct stat *statp;
{
    struct passwd *pw;

    if (lockers_name != NULL)
        free (lockers_name);
    if ((pw = (struct passwd *) getpwuid (statp->st_uid)) !=
        (struct passwd *) NULL)
    {
        lockers_name = xstrdup (pw->pw_name);
    }
    else
    {
        lockers_name = xmalloc (20);
        (void) sprintf (lockers_name, "uid%lu", (unsigned long) statp->st_uid);
    }
}

/*
 * Persistently tries to make the directory "lckdir",, which serves as a
 * lock. If the create time on the directory is greater than CVSLCKAGE
 * seconds old, just try to remove the directory.
 */
static int
set_lock (lock, will_wait)
    struct lock *lock;
    int will_wait;
{
    int waited;
    struct stat sb;
    mode_t omask;
#ifdef CVS_FUDGELOCKS
    time_t now;
#endif

    if (masterlock != NULL)
        free (masterlock);
    masterlock = xmalloc (strlen (lock->repository) + sizeof (CVSLCK) + 10);
    (void) sprintf (masterlock, "%s/%s", lock->repository, CVSLCK);

    /*
     * Note that it is up to the callers of set_lock() to arrange for signal
     * handlers that do the appropriate things, like remove the lock
     * directory before they exit.
     */
    waited = 0;
    lock->have_lckdir = 0;
    for (;;)
    {
        int status = -1;
        omask = umask (cvsumask);
        SIG_beginCrSect ();
        if (CVS_MKDIR (masterlock, 0777) == 0)
        {
            lock->have_lckdir = 1;
            SIG_endCrSect ();
            status = L_OK;
            if (waited)
                lock_obtained (lock->repository);
            goto out;
        }
        SIG_endCrSect ();
      out:
        (void) umask (omask);
        if (status != -1)
            return status;

        if (errno != EEXIST)
        {
            error (0, errno,
                   "failed to create lock directory in repository `%s'",
                   lock->repository);
            return (L_ERROR);
        }

        /* Find out who owns the lock.  If the lock directory is
           non-existent, re-try the loop since someone probably just
           removed it (thus releasing the lock).  */
        if (CVS_STAT (masterlock, &sb) < 0)
        {
            if (existence_error (errno))
                continue;

            error (0, errno, "couldn't stat lock directory `%s'", masterlock);
            return (L_ERROR);
        }
```

```
#ifdef CVS_FUDGELOCKS
        /*
         * If the create time of the directory is more than CVSLCKAGE seconds
         * ago, try to clean-up the lock directory, and if successful, just
         * quietly retry to make it.
         */
        (void) time (&now);
        if (now >= (sb.st_ctime + CVSLCKAGE))
        {
            if (CVS_RMDIR (masterlock) >= 0)
                continue;
        }
#endif

        /* set the lockers name */
        set_lockers_name (&sb);

        /* if he wasn't willing to wait, return an error */
        if (!will_wait)
            return (L_LOCKED);
        lock_wait (lock->repository);
        waited = 1;
    }
}

/*
 * Clear master lock.  We don't have to recompute the lock name since
 * clear_lock is never called except after a successful set_lock().
 */
static void
clear_lock (lock)
    struct lock *lock;
{
    SIG_beginCrSect ();
    if (CVS_RMDIR (masterlock) < 0)
        error (0, errno, "failed to remove lock dir `%s'", masterlock);
    lock->have_lckdir = 0;
    SIG_endCrSect ();
}

/*
 * Print out a message that the lock is still held, then sleep a while.
 */
static void
lock_wait (repos)
    char *repos;
{
    time_t now;

    (void) time (&now);
    error (0, 0, "[%8.8s] waiting for %s's lock in %s", ctime (&now) + 11,
           lockers_name, repos);
    /* Call cvs_flusherr to ensure that the user sees this message as
       soon as possible.  */
    cvs_flusherr ();
    (void) sleep (CVSLCKSLEEP);
}

/*
 * Print out a message when we obtain a lock.
 */
static void
lock_obtained (repos)
     char *repos;
{
    time_t now;

    (void) time (&now);
    error (0, 0, "[%8.8s] obtained lock in %s", ctime (&now) + 11, repos);
    /* Call cvs_flusherr to ensure that the user sees this message as
       soon as possible.  */
    cvs_flusherr ();
}

static int lock_filesdoneproc PROTO ((void *callerdat, int err,
                                      char *repository, char *update_dir,
                                      List *entries));

/*
 * Create a list of repositories to lock
 */
/* ARGSUSED */
static int
lock_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
```

```
{
    Node *p;

    p = getnode ();
    p->type = LOCK;
    p->key = xstrdup (repository);
    p->data = xmalloc (sizeof (struct lock));
    ((struct lock *)p->data)->repository = p->key;
    ((struct lock *)p->data)->have_lckdir = 0;

    /* FIXME-KRP: this error condition should not simply be passed by. */
    if (p->key == NULL || addnode (lock_tree_list, p) != 0)
        freenode (p);
    return (err);
}

void
lock_tree_for_write (argc, argv, local, aflag)
    int argc;
    char **argv;
    int local;
    int aflag;
{
    int err;
    /*
     * Run the recursion processor to find all the dirs to lock and lock all
     * the dirs
     */
    lock_tree_list = getlist ();
    err = start_recursion ((FILEPROC) NULL, lock_filesdoneproc,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL, argc,
                           argv, local, W_LOCAL, aflag, 0, (char *) NULL, 0);
    sortlist (lock_tree_list, fsortcmp);
    if (Writer_Lock (lock_tree_list) != 0)
        error (1, 0, "lock failed - giving up");
}

/* Lock a single directory in REPOSITORY.  It is OK to call this if
   a lock has been set with lock_dir_for_write; the new lock will replace
   the old one.  If REPOSITORY is NULL, don't do anything.  */
void
lock_dir_for_write (repository)
     char *repository;
{
    if (repository != NULL
        && (locked_dir == NULL
            || strcmp (locked_dir, repository) != 0))
    {
        Node *node;

        if (locked_dir != NULL)
            Lock_Cleanup ();

        locked_dir = xstrdup (repository);
        locked_list = getlist ();
        node = getnode ();
        node->type = LOCK;
        node->key = xstrdup (repository);
        node->data = xmalloc (sizeof (struct lock));
        ((struct lock *)node->data)->repository = node->key;
        ((struct lock *)node->data)->have_lckdir = 0;

        (void) addnode (locked_list, node);
        Writer_Lock (locked_list);
    }
}
```

A.33 log.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Print Log Information
 *
 * This line exists solely to test some pcl-cvs/ChangeLog stuff.  You
 * can delete it, if indeed it's still here when you read it.  -Karl
 *
 * Prints the RCS "log" (rlog) information for the specified files.  With no
 * argument, prints the log information for all the files in the directory
 * (recursive by default).
 */

#include "cvs.h"

/* This structure holds information parsed from the -r option.  */

struct option_revlist
{
    /* The next -r option.  */
    struct option_revlist *next;
    /* The first revision to print.  This is NULL if the range is
       :rev, or if no revision is given.  */
    char *first;
    /* The last revision to print.  This is NULL if the range is rev:,
       or if no revision is given.  If there is no colon, first and
       last are the same.  */
    char *last;
    /* Nonzero if there was a trailing '.', which means to print only
       the head revision of a branch.  */
    int branchhead;
};

/* This structure holds information derived from option_revlist given
   a particular RCS file.  */

struct revlist
{
    /* The next pair.  */
    struct revlist *next;
    /* The first numeric revision to print.  */
    char *first;
    /* The last numeric revision to print.  */
    char *last;
    /* The number of fields in these revisions (one more than
       numdots).  */
    int fields;
};

/* This structure holds information parsed from the -d option.  */

struct datelist
{
    /* The next date.  */
    struct datelist *next;
    /* The starting date.  */
    char *start;
    /* The ending date.  */
    char *end;
    /* Nonzero if the range is inclusive rather than exclusive.  */
    int inclusive;
};

/* This structure is used to pass information through start_recursion.  */
struct log_data
{
    /* Nonzero if the -R option was given, meaning that only the name
       of the RCS file should be printed.  */
    int nameonly;
    /* Nonzero if the -h option was given, meaning that only header
       information should be printed.  */
    int header;
    /* Nonzero if the -t option was given, meaning that only the
       header and the descriptive text should be printed.  */
    int long_header;
    /* Nonzero if the -N option was seen, meaning that tag information
       should not be printed.  */
    int notags;
    /* Nonzero if the -b option was seen, meaning that only revisions
       on the default branch should be printed.  */
    int default_branch;
    /* If not NULL, the value given for the -r option, which lists
       sets of revisions to be printed.  */
    struct option_revlist *revlist;
```

```
    /* If not NULL, the date pairs given for the -d option, which
       select date ranges to print.  */
    struct datelist *datelist;
    /* If not NULL, the single dates given for the -d option, which
       select specific revisions to print based on a date.  */
    struct datelist *singledatelist;
    /* If not NULL, the list of states given for the -s option, which
       only prints revisions of given states.  */
    List *statelist;
    /* If not NULL, the list of login names given for the -w option,
       which only prints revisions checked in by given users.  */
    List *authorlist;
};

/* This structure is used to pass information through walklist.  */
struct log_data_and_rcs
{
    struct log_data *log_data;
    struct revlist *revlist;
    RCSNode *rcs;
};

static Dtype log_dirproc PROTO ((void *callerdat, char *dir,
                                 char *repository, char *update_dir,
                                 List *entries));
static int log_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static struct option_revlist *log_parse_revlist PROTO ((const char *));
static void log_parse_date PROTO ((struct log_data *, const char *));
static void log_parse_list PROTO ((List **, const char *));
static struct revlist *log_expand_revlist PROTO ((RCSNode *,
                                                  struct option_revlist *,
                                                  int));
static void log_free_revlist PROTO ((struct revlist *));
static int log_version_requested PROTO ((struct log_data *, struct revlist *,
                                         RCSNode *, RCSVers *));
static int log_symbol PROTO ((Node *, void *));
static int log_count PROTO ((Node *, void *));
static int log_fix_singledate PROTO ((Node *, void *));
static int log_count_print PROTO ((Node *, void *));
static void log_tree PROTO ((struct log_data *, struct revlist *,
                             RCSNode *, const char *));
static void log_abranch PROTO ((struct log_data *, struct revlist *,
                                RCSNode *, const char *));
static void log_version PROTO ((struct log_data *, struct revlist *,
                                RCSNode *, RCSVers *, int));
static int log_branch PROTO ((Node *, void *));
static int version_compare PROTO ((const char *, const char *, int));

static const char *const log_usage[] =
{
    "Usage: %s %s [-lRhtNb] [-r[revisions]] [-d dates] [-s states]\n",
    "    [-w[logins]] [files. . .]\n",
    "\t-l\tLocal directory only, no recursion.\n",
    "\t-R\tOnly print name of RCS file.\n",
    "\t-h\tOnly print header.\n",
    "\t-t\tOnly print header and descriptive text.\n",
    "\t-N\tDo not list tags.\n",
    "\t-b\tOnly list revisions on the default branch.\n",
    "\t-r[revisions]\tSpecify revision(s)s to list.\n",
    "\t-d dates\tSpecify dates (D1<D2 for range, D for latest before).\n",
    "\t-s states\tOnly list revisions with specified states.\n",
    "\t-w[logins]\tOnly list revisions checked in by specified logins.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
cvslog (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int err = 0;
    int local = 0;
    struct log_data log_data;
    struct option_revlist *rl, **prl;

    if (argc == -1)
        usage (log_usage);

    memset (&log_data, 0, sizeof log_data);

    optind = 0;
    while ((c = getopt (argc, argv, "+bd:hlNRr::s:tw::")) != -1)
    {
        switch (c)
        {
            case 'b':
                log_data.default_branch = 1;
                break;
```

```
            case 'd':
                log_parse_date (&log_data, optarg);
                break;
            case 'h':
                log_data.header = 1;
                break;
            case 'l':
                local = 1;
                break;
            case 'N':
                log_data.notags = 1;
                break;
            case 'R':
                log_data.nameonly = 1;
                break;
            case 'r':
                rl = log_parse_revlist (optarg);
                for (prl = &log_data.revlist;
                     *prl != NULL;
                     prl = &(*prl)->next)
                    ;
                *prl = rl;
                break;
            case 's':
                log_parse_list (&log_data.statelist, optarg);
                break;
            case 't':
                log_data.long_header = 1;
                break;
            case 'w':
                if (optarg != NULL)
                    log_parse_list (&log_data.authorlist, optarg);
                else
                    log_parse_list (&log_data.authorlist, getcaller ());
                break;
            case '?':
            default:
                usage (log_usage);
                break;
        }
    }

    wrap_setup ();

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        int i;

        /* We're the local client.  Fire up the remote server.  */
        start_server ();

        ign_setup ();

        for (i = 1; i < argc && argv[i][0] == '-'; i++)
          send_arg (argv[i]);

        send_file_names (argc - i, argv + i, SEND_EXPAND_WILD);
        send_files (argc - i, argv + i, local, 0, SEND_NO_CONTENTS);

        send_to_server ("log\012", 0);
        err = get_responses_and_close ();
        return err;
    }
#endif

    err = start_recursion (log_fileproc, (FILESDONEPROC) NULL, log_dirproc,
                           (DIRLEAVEPROC) NULL, (void *) &log_data,
                           argc - optind, argv + optind, local,
                           W_LOCAL | W_REPOS | W_ATTIC, 0, 1,
                           (char *) NULL, 1);
    return (err);
}

/*
 * Parse a revision list specification.
 */

static struct option_revlist *
log_parse_revlist (argstring)
    const char *argstring;
{
    char *copy;
    struct option_revlist *ret, **pr;

    /* Unfortunately, rlog accepts -r without an argument to mean that
       latest revision on the default branch, so we must support that
       for compatibility.  */
    if (argstring == NULL)
    {
```

```
        ret = (struct option_revlist *) xmalloc (sizeof *ret);
        ret->first = NULL;
        ret->last = NULL;
        ret->next = NULL;
        ret->branchhead = 0;
        return ret;
    }

    ret = NULL;
    pr = &ret;

    /* Copy the argument into memory so that we can change it.  We
       don't want to change the argument because, at least as of this
       writing, we will use it if we send the arguments to the server.
       We never bother to free up our copy.  */
    copy = xstrdup (argstring);
    while (copy != NULL)
    {
        char *comma;
        char *cp;
        char *first, *last;
        struct option_revlist *r;

        comma = strchr (copy, ',');
        if (comma != NULL)
            *comma++ = '\0';

        first = copy;
        cp = strchr (copy, ':');
        if (cp == NULL)
            last = copy;
        else
        {
            *cp++ = '\0';
            last = cp;
        }

        if (*first == '\0')
            first = NULL;
        if (*last == '\0')
            last = NULL;

        r = (struct option_revlist *) xmalloc (sizeof *r);
        r->next = NULL;
        r->first = first;
        r->last = last;
        if (first != last
            || first[strlen (first) - 1] != '.')
        {
            r->branchhead = 0;
        }
        else
        {
            r->branchhead = 1;
            first[strlen (first) - 1] = '\0';
        }

        *pr = r;
        pr = &r->next;

        copy = comma;
    }

    return ret;
}

/*
 * Parse a date specification.
 */
static void
log_parse_date (log_data, argstring)
    struct log_data *log_data;
    const char *argstring;
{
    char *orig_copy, *copy;

    /* Copy the argument into memory so that we can change it.  We
       don't want to change the argument because, at least as of this
       writing, we will use it if we send the arguments to the server.  */
    copy = xstrdup (argstring);
    orig_copy = copy;
    while (copy != NULL)
    {
        struct datelist *nd, **pd;
        char *cpend, *cp, *ds, *de;

        nd = (struct datelist *) xmalloc (sizeof *nd);

        cpend = strchr (copy, ';');
        if (cpend != NULL)
```

```
            *cpend++ = '\0';

        pd = &log_data->datelist;
        nd->inclusive = 0;

        if ((cp = strchr (copy, '>')) != NULL)
        {
            *cp++ = '\0';
            if (*cp == '=')
            {
                ++cp;
                nd->inclusive = 1;
            }
            ds = cp;
            de = copy;
        }
        else if ((cp = strchr (copy, '<')) != NULL)
        {
            *cp++ = '\0';
            if (*cp == '=')
            {
                ++cp;
                nd->inclusive = 1;
            }
            ds = copy;
            de = cp;
        }
        else
        {
            ds = NULL;
            de = copy;
            pd = &log_data->singledatelist;
        }

        if (ds == NULL)
            nd->start = NULL;
        else if (*ds != '\0')
            nd->start = Make_Date (ds);
        else
        {
          /* 1970 was the beginning of time, as far as get_date and
             Make_Date are concerned.  FIXME: That is true only if time_t
             is a POSIX-style time and there is nothing in ANSI that
             mandates that.  It would be cleaner to set a flag saying
             whether or not there is a start date.  */
            nd->start = Make_Date ("1/1/1970 UTC");
        }

        if (*de != '\0')
            nd->end = Make_Date (de);
        else
        {
            /* We want to set the end date to some time sufficiently far
               in the future to pick up all revisions that have been
               created since the specified date and the time 'cvs log'
               completes.  FIXME: The date in question only makes sense
               if time_t is a POSIX-style time and it is 32 bits
               and signed.  We should instead be setting a flag saying
               whether or not there is an end date.  Note that using
               something like "next week" would break the testsuite (and,
               perhaps less importantly, loses if the clock is set grossly
               wrong).  */
            nd->end = Make_Date ("2038-01-01");
        }

        nd->next = *pd;
        *pd = nd;

        copy = cpend;
    }

    free (orig_copy);
}

/*
 * Parse a comma separated list of items, and add each one to *PLIST.
 */
static void
log_parse_list (plist, argstring)
    List **plist;
    const char *argstring;
{
    while (1)
    {
        Node *p;
        char *cp;

        p = getnode ();

        cp = strchr (argstring, ',');
```

```
        if (cp == NULL)
            p->key = xstrdup (argstring);
        else
        {
            size_t len;

            len = cp - argstring;
            p->key = xmalloc (len + 1);
            strncpy (p->key, argstring, len);
            p->key[len + 1] = '\0';
        }

        if (*plist == NULL)
            *plist = getlist ();
        if (addnode (*plist, p) != 0)
            freenode (p);

        if (cp == NULL)
            break;

        argstring = cp + 1;
    }
}

static int printlock_proc PROTO ((Node *, void *));

static int
printlock_proc (lock, foo)
    Node *lock;
    void *foo;
{
    cvs_output ("\n\t", 2);
    cvs_output (lock->data, 0);
    cvs_output (": ", 2);
    cvs_output (lock->key, 0);
    return 0;
}

/*
 * Do an rlog on a file
 */
static int
log_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    struct log_data *log_data = (struct log_data *) callerdat;
    Node *p;
    RCSNode *rcsfile;
    char buf[50];
    struct revlist *revlist;
    struct log_data_and_rcs log_data_and_rcs;

    if ((rcsfile = finfo->rcs) == NULL)
    {
        /* no rcs file.  What *do* we know about this file? */
        p = findnode (finfo->entries, finfo->file);
        if (p != NULL)
        {
            Entnode *e;

            e = (Entnode *) p->data;
            if (e->version[0] == '0' && e->version[1] == '\0')
            {
                if (!really_quiet)
                    error (0, 0, "%s has been added, but not committed",
                           finfo->file);
                return(0);
            }
        }

        if (!really_quiet)
            error (0, 0, "nothing known about %s", finfo->file);

        return (1);
    }

    if (log_data->nameonly)
    {
        cvs_output (rcsfile->path, 0);
        cvs_output ("\n", 1);
        return 0;
    }

    /* We will need all the information in the RCS file.  */
    RCS_fully_parse (rcsfile);

    /* Turn any symbolic revisions in the revision list into numeric
       revisions.  */
    revlist = log_expand_revlist (rcsfile, log_data->revlist,
```

```
                                log_data->default_branch);

    /* The output here is intended to be exactly compatible with the
       output of rlog.  I'm not sure whether this code should be here
       or in rcs.c; I put it here because it is specific to the log
       function, even though it uses information gathered by the
       functions in rcs.c.  */

    cvs_output ("\n", 1);

    cvs_output ("RCS file: ", 0);
    cvs_output (rcsfile->path, 0);

    cvs_output ("\nWorking file: ", 0);
    if (finfo->update_dir[0] == '\0')
        cvs_output (finfo->file, 0);
    else
    {
        cvs_output (finfo->update_dir, 0);
        cvs_output ("/", 0);
        cvs_output (finfo->file, 0);

    }

    cvs_output ("\nhead:", 0);
    if (rcsfile->head != NULL)
    {
        cvs_output (" ", 1);
        cvs_output (rcsfile->head, 0);
    }

    cvs_output ("\nbranch:", 0);
    if (rcsfile->branch != NULL)
    {
        cvs_output (" ", 1);
        cvs_output (rcsfile->branch, 0);
    }

    cvs_output ("\nlocks:", 0);
    if (rcsfile->strict_locks)
        cvs_output (" strict", 0);
    walklist (RCS_getlocks (rcsfile), printlock_proc, NULL);

    cvs_output ("\naccess list:", 0);
    if (rcsfile->access != NULL)
    {
        const char *cp;

        cp = rcsfile->access;
        while (*cp != '\0')
        {
                const char *cp2;

                cvs_output ("\n\t", 2);
                cp2 = cp;
                while (! isspace (*cp2) && *cp2 != '\0')
                    ++cp2;
                cvs_output (cp, cp2 - cp);
                cp = cp2;
                while (isspace (*cp) && *cp != '\0')
                    ++cp;
        }
    }

    if (! log_data->notags)
    {
        List *syms;

        cvs_output ("\nsymbolic names:", 0);
        syms = RCS_symbols (rcsfile);
        walklist (syms, log_symbol, NULL);
    }

    cvs_output ("\nkeyword substitution: ", 0);
    if (rcsfile->expand == NULL)
        cvs_output ("kv", 2);
    else
        cvs_output (rcsfile->expand, 0);

    cvs_output ("\ntotal revisions: ", 0);
    sprintf (buf, "%d", walklist (rcsfile->versions, log_count, NULL));
    cvs_output (buf, 0);

    if (! log_data->header && ! log_data->long_header)
    {
        cvs_output (";\tselected revisions: ", 0);

        log_data_and_rcs.log_data = log_data;
        log_data_and_rcs.revlist = revlist;
        log_data_and_rcs.rcs = rcsfile;
```

```
630         /* If any single dates were specified, we need to identify the
               revisions they select.  Each one selects the single
               revision, which is otherwise selected, of that date or
               earlier.  The log_fix_singledate routine will fill in the
               start date for each specific revision.  */
            if (log_data->singledatelist != NULL)
                walklist (rcsfile->versions, log_fix_singledate,
                          (void *) &log_data_and_rcs);

            sprintf (buf, "%d", walklist (rcsfile->versions, log_count_print,
640                                       (void *) &log_data_and_rcs));
            cvs_output (buf, 0);
        }

        cvs_output ("\n", 1);

        if (! log_data->header || log_data->long_header)
        {
            cvs_output ("description:\n", 0);
            if (rcsfile->desc != NULL)
650             cvs_output (rcsfile->desc, 0);
        }

        if (! log_data->header && ! log_data->long_header && rcsfile->head != NULL)
        {
            p = findnode (rcsfile->versions, rcsfile->head);
            if (p == NULL)
                error (1, 0, "can not find head revision in `%s'",
                       finfo->fullname);
            while (p != NULL)
660         {
                RCSVers *vers;

                vers = (RCSVers *) p->data;
                log_version (log_data, revlist, rcsfile, vers, 1);
                if (vers->next == NULL)
                    p = NULL;
                else
                {
                    p = findnode (rcsfile->versions, vers->next);
670                 if (p == NULL)
                        error (1, 0, "can not find next revision `%s' in `%s'",
                               vers->next, finfo->fullname);
                }
            }

            log_tree (log_data, revlist, rcsfile, rcsfile->head);
        }

        cvs_output("\
680 ============================================================================\n",
                   0);

        /* Free up the new revlist and restore the old one.  */
        log_free_revlist (revlist);

        /* If singledatelist is not NULL, free up the start dates we added
           to it.  */
        if (log_data->singledatelist != NULL)
        {
690         struct datelist *d;

            for (d = log_data->singledatelist; d != NULL; d = d->next)
            {
                if (d->start != NULL)
                    free (d->start);
                d->start = NULL;
            }
        }

700     return 0;
    }

    /*
     * Fix up a revision list in order to compare it against versions.
     * Expand any symbolic revisions.
     */
    static struct revlist *
    log_expand_revlist (rcs, revlist, default_branch)
        RCSNode *rcs;
710     struct option_revlist *revlist;
        int default_branch;
    {
        struct option_revlist *r;
        struct revlist *ret, **pr;

        ret = NULL;
        pr = &ret;
        for (r = revlist; r != NULL; r = r->next)
```

```
    {
720     struct revlist *nr;

        nr = (struct revlist *) xmalloc (sizeof *nr);

        if (r->first == NULL && r->last == NULL)
        {
            /* If both first and last are NULL, it means that we want
               just the head of the default branch, which is RCS_head.  */
            nr->first = RCS_head (rcs);
            nr->last = xstrdup (nr->first);
730         nr->fields = numdots (nr->first) + 1;
        }
        else if (r->branchhead)
        {
            char *branch;

            /* Print just the head of the branch.  */
            if (isdigit (r->first[0]))
                nr->first = RCS_getbranch (rcs, r->first, 1);
            else
740         {
                branch = RCS_whatbranch (rcs, r->first);
                if (branch == NULL)
                {
                    error (0, 0, "warning: `%s' is not a branch in `%s'",
                           r->first, rcs->path);
                    free (nr);
                    continue;
                }
                nr->first = RCS_getbranch (rcs, branch, 1);
750             free (branch);
            }
            if (nr->first == NULL)
            {
                error (0, 0, "warning: no revision `%s' in `%s'",
                       r->first, rcs->path);
                free (nr);
                continue;
            }
            nr->last = xstrdup (nr->first);
760         nr->fields = numdots (nr->first) + 1;
        }
        else
        {
            if (r->first == NULL || isdigit (r->first[0]))
                nr->first = xstrdup (r->first);
            else
            {
                if (RCS_nodeisbranch (rcs, r->first))
                    nr->first = RCS_whatbranch (rcs, r->first);
770             else
                    nr->first = RCS_gettag (rcs, r->first, 1, (int *) NULL);
                if (nr->first == NULL)
                {
                    error (0, 0, "warning: no revision `%s' in `%s'",
                           r->first, rcs->path);
                    free (nr);
                    continue;
                }
            }
780
            if (r->last == r->first)
                nr->last = xstrdup (nr->first);
            else if (r->last == NULL || isdigit (r->last[0]))
                nr->last = xstrdup (r->last);
            else
            {
                if (RCS_nodeisbranch (rcs, r->last))
                    nr->last = RCS_whatbranch (rcs, r->last);
                else
790                 nr->last = RCS_gettag (rcs, r->last, 1, (int *) NULL);
                if (nr->last == NULL)
                {
                    error (0, 0, "warning: no revision `%s' in `%s'",
                           r->last, rcs->path);
                    if (nr->first != NULL)
                        free (nr->first);
                    free (nr);
                    continue;
                }
800         }

            /* Process the revision numbers the same way that rlog
               does.  This code is a bit cryptic for my tastes, but
               keeping the same implementation as rlog ensures a
               certain degree of compatibility.  */
            if (r->first == NULL)
            {
                nr->fields = numdots (nr->last) + 1;
```

```
            if (nr->fields < 2)
                nr->first = xstrdup (".0");
            else
            {
                char *cp;

                nr->first = xstrdup (nr->last);
                cp = strrchr (nr->first, '.');
                strcpy (cp, ".0");
            }
        }
        else if (r->last == NULL)
        {
            nr->fields = numdots (nr->first) + 1;
            nr->last = xstrdup (nr->first);
            if (nr->fields < 2)
                nr->last[0] = '\0';
            else
            {
                char *cp;

                cp = strrchr (nr->last, '.');
                *cp = '\0';
            }
        }
        else
        {
            nr->fields = numdots (nr->first) + 1;
            if (nr->fields != numdots (nr->last) + 1
                || (nr->fields > 2
                    && version_compare (nr->first, nr->last,
                                        nr->fields - 1) != 0))
            {
                error (0, 0,
                       "invalid branch or revision pair %s:%s in '%s'",
                       r->first, r->last, rcs->path);
                free (nr->first);
                free (nr->last);
                free (nr);
                continue;
            }
            if (version_compare (nr->first, nr->last, nr->fields) > 0)
            {
                char *tmp;

                tmp = nr->first;
                nr->first = nr->last;
                nr->last = tmp;
            }
        }

        nr->next = NULL;
        *pr = nr;
        pr = &nr->next;
    }

    /* If the default branch was requested, add a revlist entry for
       it.  This is how rlog handles this option.  */
    if (default_branch
        && (rcs->head != NULL || rcs->branch != NULL))
    {
        struct revlist *nr;

        nr = (struct revlist *) xmalloc (sizeof *nr);
        if (rcs->branch != NULL)
            nr->first = xstrdup (rcs->branch);
        else
        {
            char *cp;

            nr->first = xstrdup (rcs->head);
            cp = strrchr (nr->first, '.');
            *cp = '\0';
        }
        nr->last = xstrdup (nr->first);
        nr->fields = numdots (nr->first) + 1;

        nr->next = NULL;
        *pr = nr;
    }

    return ret;
}

/*
 * Free a revlist created by log_expand_revlist.
 */
static void
log_free_revlist (revlist)
```

```
    struct revlist *revlist;
{
    struct revlist *r;

    r = revlist;
    while (r != NULL)
    {
        struct revlist *next;

        if (r->first != NULL)
            free (r->first);
        if (r->last != NULL)
            free (r->last);
        next = r->next;
        free (r);
        r = next;
    }
}

/*
 * Return nonzero if a revision should be printed, based on the
 * options provided.
 */
static int
log_version_requested (log_data, revlist, rcs, vnode)
    struct log_data *log_data;
    struct revlist *revlist;
    RCSNode *rcs;
    RCSVers *vnode;
{
    /* Handle the list of states from the -s option.  */
    if (log_data->statelist != NULL
        && findnode (log_data->statelist, vnode->state) == NULL)
    {
        return 0;
    }

    /* Handle the list of authors from the -w option.  */
    if (log_data->authorlist != NULL)
    {
        if (vnode->author != NULL
            && findnode (log_data->authorlist, vnode->author) == NULL)
        {
            return 0;
        }
    }

    /* rlog considers all the -d options together when it decides
       whether to print a revision, so we must be compatible.  */
    if (log_data->datelist != NULL || log_data->singledatelist != NULL)
    {
        struct datelist *d;

        for (d = log_data->datelist; d != NULL; d = d->next)
        {
            int cmp;

            cmp = RCS_datecmp (vnode->date, d->start);
            if (cmp > 0 || (cmp == 0 && d->inclusive))
            {
                cmp = RCS_datecmp (vnode->date, d->end);
                if (cmp < 0 || (cmp == 0 && d->inclusive))
                    break;
            }
        }

        if (d == NULL)
        {
            /* Look through the list of specific dates.  We want to
               select the revision with the exact date found in the
               start field.  The commit code ensures that it is
               impossible to check in multiple revisions of a single
               file in a single second, so checking the date this way
               should never select more than one revision.  */
            for (d = log_data->singledatelist; d != NULL; d = d->next)
            {
                if (d->start != NULL
                    && RCS_datecmp (vnode->date, d->start) == 0)
                {
                    break;
                }
            }

            if (d == NULL)
                return 0;
        }
    }

    /* If the -r or -b options were used, REVLIST will be non NULL,
       and we print the union of the specified revisions.  */
```

```
    if (revlist != NULL)
990 {
        char *v;
        int vfields;
        struct revlist *r;

        /* This code is taken from rlog.  */
        v = vnode->version;
        vfields = numdots (v) + 1;
        for (r = revlist; r != NULL; r = r->next)
        {
1000        if (vfields == r->fields + (r->fields & 1)
                && version_compare (v, r->first, r->fields) >= 0
                && version_compare (v, r->last, r->fields) <= 0)
            {
                return 1;
            }
        }

        /* If we get here, then the -b and/or the -r option was used,
           but did not match this revision, so we reject it.  */
1010
        return 0;
    }

    /* By default, we print all revisions.  */
    return 1;
}

/*
 * Output a single symbol.  This is called via walklist.
1020 */
/*ARGSUSED*/
static int
log_symbol (p, closure)
    Node *p;
    void *closure;
{
    cvs_output ("\n\t", 2);
    cvs_output (p->key, 0);
    cvs_output (": ", 2);
1030 cvs_output (p->data, 0);
    return 0;
}

/*
 * Count the number of entries on a list.  This is called via walklist.
 */
/*ARGSUSED*/
static int
log_count (p, closure)
1040 Node *p;
    void *closure;
{
    return 1;
}

/*
 * Sort out a single date specification by narrowing down the date
 * until we find the specific selected revision.
 */
1050 static int
log_fix_singledate (p, closure)
    Node *p;
    void *closure;
{
    struct log_data_and_rcs *data = (struct log_data_and_rcs *) closure;
    Node *pv;
    RCSVers *vnode;
    struct datelist *holdsingle, *holddate;
    int requested;
1060
    pv = findnode (data->rcs->versions, p->key);
    if (pv == NULL)
        error (1, 0, "missing version `%s' in RCS file `%s'",
               p->key, data->rcs->path);
    vnode = (RCSVers *) pv->data;

    /* We are only interested if this revision passes any other tests.
       Temporarily clear log_data->singledatelist to avoid confusing
       log_version_requested.  We also clear log_data->datelist,
1070   because rlog considers all the -d options together.  We don't
       want to reject a revision because it does not match a date pair
       if we are going to select it on the basis of the singledate.  */
    holdsingle = data->log_data->singledatelist;
    data->log_data->singledatelist = NULL;
    holddate = data->log_data->datelist;
    data->log_data->datelist = NULL;
    requested = log_version_requested (data->log_data, data->revlist,
                                       data->rcs, vnode);
```

```
    data->log_data->singledatelist = holdsingle;
    data->log_data->datelist = holddate;

    if (requested)
    {
        struct datelist *d;

        /* For each single date, if this revision is before the
           specified date, but is closer than the previously selected
           revision, select it instead.  */
        for (d = data->log_data->singledatelist; d != NULL; d = d->next)
        {
            if (RCS_datecmp (vnode->date, d->end) <= 0
                && (d->start == NULL
                    || RCS_datecmp (vnode->date, d->start) > 0))
            {
                if (d->start != NULL)
                    free (d->start);
                d->start = xstrdup (vnode->date);
            }
        }
    }

    return 0;
}

/*
 * Count the number of revisions we are going to print.
 */
static int
log_count_print (p, closure)
    Node *p;
    void *closure;
{
    struct log_data_and_rcs *data = (struct log_data_and_rcs *) closure;
    Node *pv;

    pv = findnode (data->rcs->versions, p->key);
    if (pv == NULL)
        error (1, 0, "missing version `%s' in RCS file `%s'",
               p->key, data->rcs->path);
    if (log_version_requested (data->log_data, data->revlist, data->rcs,
                               (RCSVers *) pv->data))
        return 1;
    else
        return 0;
}

/*
 * Print the list of changes, not including the trunk, in reverse
 * order for each branch.
 */
static void
log_tree (log_data, revlist, rcs, ver)
    struct log_data *log_data;
    struct revlist *revlist;
    RCSNode *rcs;
    const char *ver;
{
    Node *p;
    RCSVers *vnode;

    p = findnode (rcs->versions, ver);
    if (p == NULL)
        error (1, 0, "missing version `%s' in RCS file `%s'",
               ver, rcs->path);
    vnode = (RCSVers *) p->data;
    if (vnode->next != NULL)
        log_tree (log_data, revlist, rcs, vnode->next);
    if (vnode->branches != NULL)
    {
        Node *head, *branch;

        /* We need to do the branches in reverse order.  This breaks
           the List abstraction, but so does most of the branch
           manipulation in rcs.c.  */
        head = vnode->branches->list;
        for (branch = head->prev; branch != head; branch = branch->prev)
        {
            log_abranch (log_data, revlist, rcs, branch->key);
            log_tree (log_data, revlist, rcs, branch->key);
        }
    }
}

/*
 * Log the changes for a branch, in reverse order.
 */
static void
log_abranch (log_data, revlist, rcs, ver)
```

```
    struct log_data *log_data;
1170    struct revlist *revlist;
    RCSNode *rcs;
    const char *ver;
{
    Node *p;
    RCSVers *vnode;

    p = findnode (rcs->versions, ver);
    if (p == NULL)
        error (1, 0, "missing version `%s' in RCS file `%s'",
1180           ver, rcs->path);
    vnode = (RCSVers *) p->data;
    if (vnode->next != NULL)
        log_abranch (log_data, revlist, rcs, vnode->next);
    log_version (log_data, revlist, rcs, vnode, 0);
}

/*
 * Print the log output for a single version.
 */
1190 static void
log_version (log_data, revlist, rcs, ver, trunk)
    struct log_data *log_data;
    struct revlist *revlist;
    RCSNode *rcs;
    RCSVers *ver;
    int trunk;
{
    Node *p;
    int year, mon, mday, hour, min, sec;
1200    char buf[100];
    Node *padd, *pdel;

    if (! log_version_requested (log_data, revlist, rcs, ver))
        return;

    cvs_output ("----------------------------\nrevision ", 0);
    cvs_output (ver->version, 0);

    p = findnode (RCS_getlocks (rcs), ver->version);
1210    if (p != NULL)
    {
        cvs_output ("\tlocked by: ", 0);
        cvs_output (p->data, 0);
        cvs_output (";", 1);
    }

    cvs_output ("\ndate: ", 0);
    (void) sscanf (ver->date, SDATEFORM, &year, &mon, &mday, &hour, &min,
                   &sec);
1220    if (year < 1900)
        year += 1900;
    sprintf (buf, "%04d/%02d/%02d %02d:%02d:%02d", year, mon, mday,
             hour, min, sec);
    cvs_output (buf, 0);

    cvs_output (";  author: ", 0);
    cvs_output (ver->author, 0);

    cvs_output (";  state: ", 0);
1230    cvs_output (ver->state, 0);
    cvs_output (";", 1);

    if (! trunk)
    {
        padd = findnode (ver->other, ";add");
        pdel = findnode (ver->other, ";delete");
    }
    else if (ver->next == NULL)
    {
1240        padd = NULL;
        pdel = NULL;
    }
    else
    {
        Node *nextp;
        RCSVers *nextver;

        nextp = findnode (rcs->versions, ver->next);
        if (nextp == NULL)
1250            error (1, 0, "missing version `%s' in `%s'", ver->next,
                   rcs->path);
        nextver = (RCSVers *) nextp->data;
        pdel = findnode (nextver->other, ";add");
        padd = findnode (nextver->other, ";delete");
    }

    if (padd != NULL)
    {
```

```
            cvs_output ("  lines: +", 0);
            cvs_output (padd->data, 0);
            cvs_output (" -", 2);
            cvs_output (pdel->data, 0);
        }

        if (ver->branches != NULL)
        {
            cvs_output ("\nbranches:", 0);
            walklist (ver->branches, log_branch, (void *) NULL);
        }

        cvs_output ("\n", 1);

        p = findnode (ver->other, "log");
        /* The p->date == NULL case is the normal one for an empty log
           message (rcs-14 in sanity.sh).  I don't think the case where
           p->data is "" can happen (getrcskey in rcs.c checks for an
           empty string and set the value to NULL in that case).  My guess
           would be the p == NULL case would mean an RCS file which was
           missing the "log" keyword (which is illegal according to
           rcsfile.5).  */
        if (p == NULL || p->data == NULL || p->data[0] == '\0')
            cvs_output ("*** empty log message ***\n", 0);
        else
        {
            /* FIXME: Technically, the log message could contain a null
               byte.  */
            cvs_output (p->data, 0);
            if (p->data[strlen (p->data) - 1] != '\n')
                cvs_output ("\n", 1);
        }
    }

    /*
     * Output a branch version.  This is called via walklist.
     */
    /*ARGSUSED*/
    static int
    log_branch (p, closure)
        Node *p;
        void *closure;
    {
        cvs_output ("  ", 2);
        if ((numdots (p->key) & 1) == 0)
            cvs_output (p->key, 0);
        else
        {
            char *f, *cp;

            f = xstrdup (p->key);
            cp = strrchr (f, '.');
            *cp = '\0';
            cvs_output (f, 0);
            free (f);
        }
        cvs_output (";", 1);
        return 0;
    }

    /*
     * Print a warm fuzzy message
     */
    /* ARGSUSED */
    static Dtype
    log_dirproc (callerdat, dir, repository, update_dir, entries)
        void *callerdat;
        char *dir;
        char *repository;
        char *update_dir;
        List *entries;
    {
        if (!isdir (dir))
            return (R_SKIP_ALL);

        if (!quiet)
            error (0, 0, "Logging %s", update_dir);
        return (R_PROCESS);
    }

    /*
     * Compare versions.  This is taken from RCS compartial.
     */
    static int
    version_compare (v1, v2, len)
        const char *v1;
        const char *v2;
        int len;
    {
        while (1)
```

```
    {
1350    int d1, d2, r;

        if (*v1 == '\0')
            return 1;
        if (*v2 == '\0')
            return -1;

        while (*v1 == '0')
            ++v1;
        for (d1 = 0; isdigit (v1[d1]); ++d1)
1360        ;

        while (*v2 == '0')
            ++v2;
        for (d2 = 0; isdigit (v2[d2]); ++d2)
            ;

        if (d1 != d2)
            return d1 < d2 ? -1 : 1;

1370    r = memcmp (v1, v2, d1);
        if (r != 0)
            return r;

        --len;
        if (len == 0)
            return 0;

        v1 += d1;
        v2 += d1;
1380
        if (*v1 == '.')
            ++v1;
        if (*v2 == '.')
            ++v2;
    }
}
```

A.34 login.c

```
/*
 * Copyright (c) 1995, Cyclic Software, Bloomington, IN, USA
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with CVS.
 *
 * Allow user to log in for an authenticating server.
 */

#include "cvs.h"
#include "getline.h"

#ifdef AUTH_CLIENT_SUPPORT    /* This covers the rest of the file. */

/* There seems to be very little agreement on which system header
   getpass is declared in.  With a lot of fancy autoconfiscation,
   we could perhaps detect this, but for now we'll just rely on
   _CRAY, since Cray is perhaps the only system on which our own
   declaration won't work (some Crays declare the 2#$% thing as
   varadic, believe it or not).  On Cray, getpass will be declared
*/
#ifndef _CRAY
extern char *getpass ();
#endif

#ifndef CVS_PASSWORD_FILE
#define CVS_PASSWORD_FILE ".cvspass"
#endif

/* If non-NULL, get_cvs_password() will just return this. */
static char *cvs_password = NULL;

static char *construct_cvspass_filename PROTO ((void));

/* The return value will need to be freed. */
static char *
construct_cvspass_filename ()
{
    char *homedir;
    char *passfile;

    /* Environment should override file. */
    if ((passfile = getenv ("CVS_PASSFILE")) != NULL)
        return xstrdup (passfile);

    /* Construct absolute pathname to user's password file. */
    /* todo: does this work under OS/2 ? */
    homedir = get_homedir ();
    if (! homedir)
    {
        error (1, errno, "could not find out home directory");
        return (char *) NULL;
    }

    passfile =
        (char *) xmalloc (strlen (homedir) + strlen (CVS_PASSWORD_FILE) + 3);
    strcpy (passfile, homedir);
#ifndef NO_SLASH_AFTER_HOME
    /* NO_SLASH_AFTER_HOME is defined for VMS, where foo:[bar]/.cvspass is not
       a legal filename but foo:[bar].cvspass is.  A more clean solution would
       be something more along the lines of a "join a directory to a filename"
       kind of thing....  */
    strcat (passfile, "/");
#endif
    strcat (passfile, CVS_PASSWORD_FILE);

    /* Safety first and last, Scouts. */
    if (isfile (passfile))
        /* xchmod() is too polite. */
        chmod (passfile, 0600);

    return passfile;
}

static const char *const login_usage[] =
{
    "Usage: %s %s\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

/* Prompt for a password, and store it in the file "CVS/.cvspass".
 *
 * Because the user might be accessing multiple repositories, with
 * different passwords for each one, the format of ~/.cvspass is:
 *
 * userhost:/path Acleartext_password
 * userhost:/path Acleartext_password
```

```
 * ...
 *
 * Of course, the "user" might be left off -- it's just based on the
 * value of CVSroot.
 *
 * The "A" before "cleartext_password" is a literal capital A.  It's a
 * version number indicating which form of scrambling we're doing on
 * the password -- someday we might provide something more secure than
 * the trivial encoding we do now, and when that day comes, it would
 * be nice to remain backward-compatible.
 *
 * Like .netrc, the file's permissions are the only thing preventing
 * it from being read by others.  Unlike .netrc, we will not be
 * fascist about it, at most issuing a warning, and never refusing to
 * work.
 */
int
login (argc, argv)
    int argc;
    char **argv;
{
    char *passfile;
    FILE *fp;
    char *typed_password, *found_password;
    char *linebuf = (char *) NULL;
    size_t linebuf_len;
    int root_len, already_entered = 0;
    int line_length;

    if (argc < 0)
        usage (login_usage);

    if (CVSroot_method != pserver_method)
    {
        error (0, 0, "can only use pserver method with 'login' command");
        error (1, 0, "CVSROOT: %s", CVSroot_original);
    }

    if (! CVSroot_username)
    {
        error (0, 0, "CVSROOT \"%s\" is not fully-qualified.",
               CVSroot_original);
        error (1, 0, "Please make sure to specify \"user@host\"!");
    }

    printf ("(Logging in to %s@%s)\n", CVSroot_username, CVSroot_hostname);
    fflush (stdout);

    passfile = construct_cvspass_filename ();
    typed_password = getpass ("CVS password: ");
    typed_password = scramble (typed_password);

    /* Force get_cvs_password() to use this one (when the client
     * confirms the new password with the server), instead of
     * consulting the file.  We make a new copy because cvs_password
     * will get zeroed by connect_to_server().  */

    cvs_password = xstrdup (typed_password);

    connect_to_server (NULL, NULL, 1, AUTH_PASSWORD);

    /* IF we have a password for this "[user@]host:/path" already
     *  THEN
     *    IF it's the same as the password we read from the prompt
     *     THEN
     *       do nothing
     *     ELSE
     *       replace the old password with the new one
     *  ELSE
     *    append new entry to the end of the file.
*/

    root_len = strlen (CVSroot_original);

    /* Yes, the method below reads the user's password file twice.  It's
       inefficient, but we're not talking about a gig of data here. */

    fp = CVS_FOPEN (passfile, "r");
    /* FIXME: should be printing a message if fp == NULL and not
       existence_error (errno).  */
    if (fp != NULL)
    {
        /* Check each line to see if we have this entry already. */
        while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
        {
          if (strncmp (CVSroot_original, linebuf, root_len) == 0)
            {
                already_entered = 1;
                break;
            }
```

```
        }
180     if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", passfile);
    }
    else if (!existence_error (errno))
        error (0, errno, "cannot open %s", passfile);

    if (already_entered)
    {
        /* This user/host has a password in the file already. */

190     strtok (linebuf, " ");
        found_password = strtok (NULL, "\n");
        if (strcmp (found_password, typed_password))
        {
            /* typed_password and found_password don't match, so we'll
             * have to update passfile.  We replace the old password
             * with the new one by writing a tmp file whose contents are
             * exactly the same as passfile except that this one entry
             * gets typed_password instead of found_password.  Then we
             * rename the tmp file on top of passfile.
200          */
            char *tmp_name;
            FILE *tmp_fp;

            tmp_name = cvs_temp_name ();
            if ((tmp_fp = CVS_FOPEN (tmp_name, "w")) == NULL)
            {
                error (1, errno, "unable to open temp file %s", tmp_name);
                return 1;
            }
210         chmod (tmp_name, 0600);

            fp = CVS_FOPEN (passfile, "r");
            if (fp == NULL)
            {
                error (1, errno, "unable to open %s", passfile);
                if (linebuf)
                    free (linebuf);
                return 1;
            }
220         /* I'm not paranoid, they really ARE out to get me: */
            chmod (passfile, 0600);

            while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
            {
                if (strncmp (CVSroot_original, linebuf, root_len))
                {
                    if (fprintf (tmp_fp, "%s", linebuf) == EOF)
                        error (0, errno, "cannot write %s", tmp_name);
                }
230             else
                {
                    if (fprintf (tmp_fp, "%s %s\n", CVSroot_original,
                                 typed_password) == EOF)
                        error (0, errno, "cannot write %s", tmp_name);
                }
            }
            if (line_length < 0 && !feof (fp))
                error (0, errno, "cannot read %s", passfile);
            if (linebuf)
240             free (linebuf);
            if (fclose (tmp_fp) < 0)
                error (0, errno, "cannot close %s", tmp_name);
            if (fclose (fp) < 0)
                error (0, errno, "cannot close %s", passfile);

            /* FIXME: rename_file would make more sense (e.g. almost
               always faster).  */
            copy_file (tmp_name, passfile);
            unlink_file (tmp_name);
250         chmod (passfile, 0600);

            free (tmp_name);
        }
    }
    else
    {
        if (linebuf)
            free (linebuf);
        if ((fp = CVS_FOPEN (passfile, "a")) == NULL)
260     {
            error (1, errno, "could not open %s", passfile);
            free (passfile);
            return 1;
        }

        if (fprintf (fp, "%s %s\n", CVSroot_original, typed_password) == EOF)
            error (0, errno, "cannot write %s", passfile);
        if (fclose (fp) < 0)
```

```
            error (0, errno, "cannot close %s", passfile);
    }

    /* Utter, total, raving paranoia, I know. */
    chmod (passfile, 0600);
    memset (typed_password, 0, strlen (typed_password));
    free (typed_password);

    free (passfile);
    free (cvs_password);
    cvs_password = NULL;
    return 0;
}

/* Returns the _scrambled_ password.  The server must descramble
   before hashing and comparing. */
char *
get_cvs_password ()
{
    int found_it = 0;
    int root_len;
    char *password;
    char *linebuf = (char *) NULL;
    size_t linebuf_len;
    FILE *fp;
    char *passfile;
    int line_length;

    /* If someone (i.e., login()) is calling connect_to_pserver() out of
       context, then assume they have supplied the correct, scrambled
       password. */
    if (cvs_password)
        return cvs_password;

    if (getenv ("CVS_PASSWORD") != NULL)
    {
        /* In previous versions of CVS one could specify a password in
           CVS_PASSWORD.  This is a bad idea, because in BSD variants
           of unix anyone can see the environment variable with 'ps'.
           But for users who were using that feature we want to at
           least let them know what is going on.  After printing this
           warning, we should fall through to the regular error where
           we tell them to run "cvs login" (unless they already ran
           it, of course).  */
        error (0, 0, "CVS_PASSWORD is no longer supported; ignored");
    }

    /* Else get it from the file.  First make sure that the CVSROOT
       variable has the appropriate fields filled in. */

    if (CVSroot_method != pserver_method)
    {
        error (0, 0, "can only call GET_CVS_PASSWORD  with pserver method");
        error (1, 0, "CVSROOT: %s", CVSroot_original);
    }

    if (! CVSroot_username)
    {
        error (0, 0, "CVSROOT \"%s\" is not fully-qualified.",
               CVSroot_original);
        error (1, 0, "Please make sure to specify \"user@host\"!");
    }

    passfile = construct_cvspass_filename ();
    fp = CVS_FOPEN (passfile, "r");
    if (fp == NULL)
    {
        error (0, errno, "could not open %s", passfile);
        free (passfile);
        error (1, 0, "use \"cvs login\" to log in first");
    }

    root_len = strlen (CVSroot_original);

    /* Check each line to see if we have this entry already. */
    while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
    {
        if (strncmp (CVSroot_original, linebuf, root_len) == 0)
        {
            /* This is it!  So break out and deal with linebuf. */
            found_it = 1;
            break;
        }
    }
    if (line_length < 0 && !feof (fp))
        error (0, errno, "cannot read %s", passfile);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", passfile);

    if (found_it)
```

```
    {
360     /* linebuf now contains the line with the password. */
        char *tmp;

        strtok (linebuf, " ");
        password = strtok (NULL, "\n");

        /* Give it permanent storage. */
        tmp = xstrdup (password);
        memset (password, 0, strlen (password));
        free (linebuf);
370     return tmp;
    }
    else
    {
        if (linebuf)
            free (linebuf);
        error (0, 0, "cannot find password");
        error (1, 0, "use \"cvs login\" to log in first");
    }
    /* NOTREACHED */
380 return NULL;
}

static const char *const logout_usage[] =
{
    "Usage: %s %s\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

390 /* Remove any entry for the CVSRoot repository found in "CVS/.cvspass". */
int
logout (argc, argv)
    int argc;
    char **argv;
{
    char *passfile;
    FILE *fp;
    char *tmp_name;
    FILE *tmp_fp;
400 char *linebuf = (char *) NULL;
    size_t linebuf_len;
    int root_len, found = 0;
    int line_length;

    if (argc < 0)
        usage (logout_usage);

    if (CVSroot_method != pserver_method)
    {
410     error (0, 0, "can only use pserver method with `logout' command");
        error (1, 0, "CVSROOT: %s", CVSroot_original);
    }

    if (! CVSroot_username)
    {
        error (0, 0, "CVSROOT \"%s\" is not fully-qualified.",
               CVSroot_original);
        error (1, 0, "Please make sure to specify \"user@host\"!");
    }
420
    /* Hmm.  Do we want a variant of this command which deletes _all_
       the entries from the current .cvspass?  Might be easier to
       remember than "rm ~/.cvspass" but then again if people are
       mucking with HOME (common in Win95 as the system doesn't set
       it), then this variant of "cvs logout" might give a false sense
       of security, in that it wouldn't delete entries from any
       .cvspass files but the current one.  */

    printf ("(Logging out of %s@%s)\n", CVSroot_username, CVSroot_hostname);
430 fflush (stdout);

    /* IF we have a password for this "[user]host:/path" already
     *  THEN
     *    drop the entry
     *  ELSE
     *    do nothing
     */

    passfile = construct_cvspass_filename ();
440 tmp_name = cvs_temp_name ();
    if ((tmp_fp = CVS_FOPEN (tmp_name, "w")) == NULL)
    {
        error (1, errno, "unable to open temp file %s", tmp_name);
        return 1;
    }
    chmod (tmp_name, 0600);

    root_len = strlen (CVSroot_original);
```

```
    fp = CVS_FOPEN (passfile, "r");
    if (fp == NULL)
        error (1, errno, "Error opening %s", passfile);

    /* Check each line to see if we have this entry. */
    /* Copy only those lines that do not match this entry */
    while ((line_length = getline (&linebuf, &linebuf_len, fp)) >= 0)
    {
        if (strncmp (CVSroot_original, linebuf, root_len))
        {
            if (fprintf (tmp_fp, "%s", linebuf) == EOF)
                error (0, errno, "cannot write %s", tmp_name);
        }
        else
            found = 1;
    }
    if (line_length < 0 && !feof (fp))
        error (0, errno, "cannot read %s", passfile);

    if (linebuf)
        free (linebuf);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", passfile);
    if (fclose (tmp_fp) < 0)
        error (0, errno, "cannot close %s", tmp_name);

    if (! found)
    {
        printf ("Entry not found for %s\n", CVSroot_original);
        unlink_file (tmp_name);
    }
    else
    {
        /* FIXME: rename_file would make more sense (e.g. almost
           always faster).  */
        copy_file (tmp_name, passfile);
        unlink_file (tmp_name);
        chmod (passfile, 0600);
    }
    return 0;
}

#endif /* AUTH_CLIENT_SUPPORT from beginning of file. */
```

A.35 logmsg.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"
#include "getline.h"

static int find_type PROTO((Node * p, void *closure));
static int fmt_proc PROTO((Node * p, void *closure));
static int logfile_write PROTO((char *repository, char *filter,
                        char *message, FILE * logfp, List * changes));
static int rcsinfo_proc PROTO((char *repository, char *template));
static int title_proc PROTO((Node * p, void *closure));
static int update_logfile_proc PROTO((char *repository, char *filter));
static void setup_tmpfile PROTO((FILE * xfp, char *xprefix, List * changes));
static int editinfo_proc PROTO((char *repository, char *template));
static int verifymsg_proc PROTO((char *repository, char *script));

static FILE *fp;
static char *str_list;
static char *str_list_format; /* The format for str_list's contents. */
static char *editinfo_editor;
static char *verifymsg_script;
static Ctype type;

/*
 * Puts a standard header on the output which is either being prepared for an
 * editor session, or being sent to a logfile program.  The modified, added,
 * and removed files are included (if any) and formatted to look pretty. */
static char *prefix;
static int col;
static char *tag;
static void
setup_tmpfile (xfp, xprefix, changes)
    FILE *xfp;
    char *xprefix;
    List *changes;
{
    /* set up statics */
    fp = xfp;
    prefix = xprefix;

    type = T_MODIFIED;
    if (walklist (changes, find_type, NULL) != 0)
    {
        (void) fprintf (fp, "%sModified Files:\n", prefix);
        col = 0;
        (void) walklist (changes, fmt_proc, NULL);
        (void) fprintf (fp, "\n");
        if (tag != NULL)
        {
            free (tag);
            tag = NULL;
        }
    }
    type = T_ADDED;
    if (walklist (changes, find_type, NULL) != 0)
    {
        (void) fprintf (fp, "%sAdded Files:\n", prefix);
        col = 0;
        (void) walklist (changes, fmt_proc, NULL);
        (void) fprintf (fp, "\n");
        if (tag != NULL)
        {
            free (tag);
            tag = NULL;
        }
    }
    type = T_REMOVED;
    if (walklist (changes, find_type, NULL) != 0)
    {
        (void) fprintf (fp, "%sRemoved Files:\n", prefix);
        col = 0;
        (void) walklist (changes, fmt_proc, NULL);
        (void) fprintf (fp, "\n");
        if (tag != NULL)
        {
            free (tag);
            tag = NULL;
        }
    }
}

/*
```

```
 * Looks for nodes of a specified type and returns 1 if found
 */
static int
find_type (p, closure)
    Node *p;
    void *closure;
{
    struct logfile_info *li;

    li = (struct logfile_info *) p->data;
    if (li->type == type)
        return (1);
    else
        return (0);
}

/*
 * Breaks the files list into reasonable sized lines to avoid line wrap...
 * all in the name of pretty output.  It only works on nodes whose types
 * match the one we're looking for
 */
static int
fmt_proc (p, closure)
    Node *p;
    void *closure;
{
    struct logfile_info *li;

    li = (struct logfile_info *) p->data;
    if (li->type == type)
    {
        if (li->tag == NULL
            ? tag != NULL
            : tag == NULL || strcmp (tag, li->tag) != 0)
        {
            if (col > 0)
                (void) fprintf (fp, "\n");
            (void) fprintf (fp, "%s", prefix);
            col = strlen (prefix);
            while (col < 6)
            {
                (void) fprintf (fp, " ");
                ++col;
            }

            if (li->tag == NULL)
                (void) fprintf (fp, "No tag");
            else
                (void) fprintf (fp, "Tag: %s", li->tag);

            if (tag != NULL)
                free (tag);
            tag = xstrdup (li->tag);

            /* Force a new line.  */
            col = 70;
        }

        if (col == 0)
        {
            (void) fprintf (fp, "%s\t", prefix);
            col = 8;
        }
        else if (col > 8 && (col + (int) strlen (p->key)) > 70)
        {
            (void) fprintf (fp, "\n%s\t", prefix);
            col = 8;
        }
        (void) fprintf (fp, "%s ", p->key);
        col += strlen (p->key) + 1;
    }
    return (0);
}

/*
 * Builds a temporary file using setup_tmpfile() and invokes the user's
 * editor on the file.  The header garbage in the resultant file is then
 * stripped and the log message is stored in the "message" argument.
 *
 * If REPOSITORY is non-NULL, process rcsinfo for that repository; if it
 * is NULL, use the CVSADM_TEMPLATE file instead.
 */
void
do_editor (dir, messagep, repository, changes)
    char *dir;
    char **messagep;
    char *repository;
    List *changes;
{
    static int reuse_log_message = 0;
```

```
    char *line;
    int line_length;
    size_t line_chars_allocated;
    char *fname;
    struct stat pre_stbuf, post_stbuf;
    int retcode = 0;
    char *p;

    if (noexec || reuse_log_message)
        return;

    /* Abort creation of temp file if no editor is defined */
    if (strcmp (Editor, "") == 0 && !editinfo_editor)
        error(1, 0, "no editor defined, must use -e or -m");


    /* Create a temporary file */
    fname = cvs_temp_name ();
again:
    if ((fp = CVS_FOPEN (fname, "w+")) == NULL)
        error (1, 0, "cannot create temporary file %s", fname);

    if (*messagep)
    {
        (void) fprintf (fp, "%s", *messagep);

        if ((*messagep)[0] == '\0' ||
            (*messagep)[strlen (*messagep) - 1] != '\n')
            (void) fprintf (fp, "\n");
    }
    else
        (void) fprintf (fp, "\n");

    if (repository != NULL)
        /* tack templates on if necessary */
        (void) Parse_Info (CVSROOTADM_RCSINFO, repository, rcsinfo_proc, 1);
    else
    {
        FILE *tfp;
        char buf[1024];
        char *p;
        size_t n;
        size_t nwrite;

        /* Why "b"?  */
        tfp = CVS_FOPEN (CVSADM_TEMPLATE, "rb");
        if (tfp == NULL)
        {
            if (!existence_error (errno))
                error (1, errno, "cannot read %s", CVSADM_TEMPLATE);
        }
        else
        {
            while (!feof (tfp))
            {
                n = fread (buf, 1, sizeof buf, tfp);
                nwrite = n;
                p = buf;
                while (nwrite > 0)
                {
                    n = fwrite (p, 1, nwrite, fp);
                    nwrite -= n;
                    p += n;
                }
                if (ferror (tfp))
                    error (1, errno, "cannot read %s", CVSADM_TEMPLATE);
            }
            if (fclose (tfp) < 0)
                error (0, errno, "cannot close %s", CVSADM_TEMPLATE);
        }
    }

    (void) fprintf (fp,
"%s----------------------------------------------------------------------\n",
                    CVSEDITPREFIX);
    (void) fprintf (fp,
"%sEnter Log.  Lines beginning with `%.*s' are removed automatically\n%s\n",
                    CVSEDITPREFIX, CVSEDITPREFIXLEN, CVSEDITPREFIX,
                    CVSEDITPREFIX);
    if (dir != NULL && *dir)
        (void) fprintf (fp, "%sCommitting in %s\n%s\n", CVSEDITPREFIX,
                        dir, CVSEDITPREFIX);
    if (changes != NULL)
        setup_tmpfile (fp, CVSEDITPREFIX, changes);
    (void) fprintf (fp,
"%s----------------------------------------------------------------------\n",
                    CVSEDITPREFIX);

    /* finish off the temp file */
    if (fclose (fp) == EOF)
```

```
        error (1, errno, "%s", fname);
    if ( CVS_STAT (fname, &pre_stbuf) == -1)
        pre_stbuf.st_mtime = 0;

    if (editinfo_editor)
        free (editinfo_editor);
    editinfo_editor = (char *) NULL;
#ifdef CLIENT_SUPPORT
    if (client_active)
        ; /* nothing, leave editinfo_editor NULL */
    else
#endif
    if (repository != NULL)
        (void) Parse_Info (CVSROOTADM_EDITINFO, repository, editinfo_proc, 0);

    /* run the editor */
    run_setup (editinfo_editor ? editinfo_editor : Editor);
    run_arg (fname);
    if ((retcode = run_exec (RUN_TTY, RUN_TTY, RUN_TTY,
                             RUN_NORMAL | RUN_SIGIGNORE)) != 0)
        error (editinfo_editor ? 1 : 0, retcode == -1 ? errno : 0,
               editinfo_editor ? "Logfile verification failed" :
               "warning: editor session failed");

    /* put the entire message back into the *messagep variable */

    fp = open_file (fname, "r");

    if (*messagep)
        free (*messagep);

    if ( CVS_STAT (fname, &post_stbuf) != 0)
            error (1, errno, "cannot find size of temp file %s", fname);

    if (post_stbuf.st_size == 0)
        *messagep = NULL;
    else
    {
        /* On NT, we might read less than st_size bytes, but we won't
           read more.  So this works.  */
        *messagep = (char *) xmalloc (post_stbuf.st_size + 1);
        *messagep[0] = '\0';
    }

    line = NULL;
    line_chars_allocated = 0;

    if (*messagep)
    {
        p = *messagep;
        while (1)
        {
            line_length = getline (&line, &line_chars_allocated, fp);
            if (line_length == -1)
            {
                if (ferror (fp))
                    error (0, errno, "warning: cannot read %s", fname);
                break;
            }
            if (strncmp (line, CVSEDITPREFIX, CVSEDITPREFIXLEN) == 0)
                continue;
            (void) strcpy (p, line);
            p += line_length;
        }
    }
    if (fclose (fp) < 0)
        error (0, errno, "warning: cannot close %s", fname);

    if (pre_stbuf.st_mtime == post_stbuf.st_mtime ||
        *messagep == NULL ||
        strcmp (*messagep, "\n") == 0)
    {
        for (;;)
        {
            (void) printf ("\nLog message unchanged or not specified\n");
            (void) printf ("a)bort, c)ontinue, e)dit, !)reuse this message unchanged for remaining dirs\n");
            (void) printf ("Action: (continue) ");
            (void) fflush (stdout);
            line_length = getline (&line, &line_chars_allocated, stdin);
            if (line_length < 0)
            {
                error (0, errno, "cannot read from stdin");
                if (unlink_file (fname) < 0)
                    error (0, errno,
                           "warning: cannot remove temp file %s", fname);
                error (1, 0, "aborting");
            }
            else if (line_length == 0
                     || *line == '\n' || *line == 'c' || *line == 'C')
                break;
```

```
            if (*line == 'a' || *line == 'A')
                {
                    if (unlink_file (fname) < 0)
                        error (0, errno, "warning: cannot remove temp file %s", fname);
                    error (1, 0, "aborted by user");
                }
            if (*line == 'e' || *line == 'E')
                goto again;
            if (*line == '!')
            {
                reuse_log_message = 1;
                break;
            }
            (void) printf ("Unknown input\n");
        }
    }
    if (line)
        free (line);
    if (unlink_file (fname) < 0)
        error (0, errno, "warning: cannot remove temp file %s", fname);
    free (fname);
}

/* Runs the user-defined verification script as part of the commit or import
   process.  This verification is meant to be run whether or not the user
   included the -m atribute.  unlike the do_editor function, this is
   independant of the running of an editor for getting a message.
 */
void
do_verify (message, repository)
    char *message;
    char *repository;
{
    FILE *fp;
    char *fname;
    int retcode = 0;

#ifdef CLIENT_SUPPORT
    if (client_active)
        /* The verification will happen on the server.  */
        return;
#endif

    /* FIXME? Do we really want to skip this on noexec?  What do we do
       for the other administrative files?  */
    if (noexec)
        return;

    /* If there's no message, then we have nothing to verify.  Can this
       case happen?  And if so why would we print a message?  */
    if (message == NULL)
    {
        cvs_output ("No message to verify\n", 0);
        return;
    }

    /* Get a temp filename, open a temporary file, write the message to the
       temp file, and close the file.  */

    fname = cvs_temp_name ();

    fp = fopen (fname, "w");
    if (fp == NULL)
        error (1, errno, "cannot create temporary file %s", fname);
    else
    {
        fprintf (fp, "%s", message);
        if ((message)[0] == '\0' ||
            (message)[strlen (message) - 1] != '\n')
            (void) fprintf (fp, "%s", "\n");
        if (fclose (fp) == EOF)
            error (1, errno, "%s", fname);

        /* Get the name of the verification script to run  */

        if (repository != NULL)
            (void) Parse_Info (CVSROOTADM_VERIFYMSG, repository,
                               verifymsg_proc, 0);

        /* Run the verification script  */

        if (verifymsg_script)
        {
            run_setup (verifymsg_script);
            run_arg (fname);
            if ((retcode = run_exec (RUN_TTY, RUN_TTY, RUN_TTY,
                                     RUN_NORMAL | RUN_SIGIGNORE)) != 0)
            {
                /* Since following error() exits, delete the temp file
                   now.  */
```

```
                unlink_file (fname);

                error (1, retcode == -1 ? errno : 0,
                       "Message verification failed");
            }
        }

        /* Delete the temp file  */

        unlink_file (fname);
        free (fname);
    }
}

/*
 * callback proc for Parse_Info for rcsinfo templates this routine basically
 * copies the matching template onto the end of the tempfile we are setting
 * up
 */
/* ARGSUSED */
static int
rcsinfo_proc (repository, template)
    char *repository;
    char *template;
{
    static char *last_template;
    FILE *tfp;

    /* nothing to do if the last one included is the same as this one */
    if (last_template && strcmp (last_template, template) == 0)
        return (0);
    if (last_template)
        free (last_template);
    last_template = xstrdup (template);

    if ((tfp = CVS_FOPEN (template, "r")) != NULL)
    {
        char *line = NULL;
        size_t line_chars_allocated = 0;

        while (getline (&line, &line_chars_allocated, tfp) >= 0)
            (void) fputs (line, fp);
        if (ferror (tfp))
            error (0, errno, "warning: cannot read %s", template);
        if (fclose (tfp) < 0)
            error (0, errno, "warning: cannot close %s", template);
        if (line)
            free (line);
        return (0);
    }
    else
    {
        error (0, errno, "Couldn't open rcsinfo template file %s", template);
        return (1);
    }
}

/*
 * Uses setup_tmpfile() to pass the updated message on directly to any
 * logfile programs that have a regular expression match for the checked in
 * directory in the source repository.  The log information is fed into the
 * specified program as standard input.
 */
static FILE *logfp;
static char *message;
static List *changes;

void
Update_Logfile (repository, xmessage, xlogfp, xchanges)
    char *repository;
    char *xmessage;
    FILE *xlogfp;
    List *xchanges;
{
    /* nothing to do if the list is empty */
    if (xchanges == NULL || xchanges->list->next == xchanges->list)
        return;

    /* set up static vars for update_logfile_proc */
    message = xmessage;
    logfp = xlogfp;
    changes = xchanges;

    /* call Parse_Info to do the actual logfile updates */
    (void) Parse_Info (CVSROOTADM_LOGINFO, repository, update_logfile_proc, 1);
}

/*
 * callback proc to actually do the logfile write from Update_Logfile
 */
```

```
static int
update_logfile_proc (repository, filter)
    char *repository;
    char *filter;
{
    return (logfile_write (repository, filter, message, logfp, changes));
}

/*
 * concatenate each filename/version onto str_list
 */
static int
title_proc (p, closure)
    Node *p;
    void *closure;
{
    struct logfile_info *li;
    char *c;

    li = (struct logfile_info *) p->data;
    if (li->type == type)
    {
        /* Until we decide on the correct logging solution when we add
           directories or perform imports, T_TITLE nodes will only
           tack on the name provided, regardless of the format string.
           You can verify that this assumption is safe by checking the
           code in add.c (add_directory) and import.c (import). */

        str_list = xrealloc (str_list, strlen (str_list) + 5);
        (void) strcat (str_list, " ");

        if (li->type == T_TITLE)
        {
            str_list = xrealloc (str_list,
                                 strlen (str_list) + strlen (p->key) + 5);
            (void) strcat (str_list, p->key);
        }
        else
        {
            /* All other nodes use the format string. */

            for (c = str_list_format; *c != '\0'; c++)
            {
                switch (*c)
                {
                case 's':
                    str_list =
                        xrealloc (str_list,
                                  strlen (str_list) + strlen (p->key) + 5);
                    (void) strcat (str_list, p->key);
                    break;
                case 'V':
                    str_list =
                        xrealloc (str_list,
                                  (strlen (str_list)
                                   + (li->rev_old ? strlen (li->rev_old) : 0)
                                   + 10)
                                  );
                    (void) strcat (str_list, (li->rev_old
                                              ? li->rev_old : "NONE"));
                    break;
                case 'v':
                    str_list =
                        xrealloc (str_list,
                                  (strlen (str_list)
                                   + (li->rev_new ? strlen (li->rev_new) : 0)
                                   + 10)
                                  );
                    (void) strcat (str_list, (li->rev_new
                                              ? li->rev_new : "NONE"));
                    break;
                /* All other characters, we insert an empty field (but
                   we do put in the comma separating it from other
                   fields).  This way if future CVS versions add formatting
                   characters, one can write a loginfo file which at least
                   won't blow up on an old CVS.  */
                }
                if (*(c + 1) != '\0')
                {
                    str_list = xrealloc (str_list, strlen (str_list) + 5);
                    (void) strcat (str_list, ",");
                }
            }
        }
    }
    return (0);
}

/*
 * Writes some stuff to the logfile "filter" and returns the status of the
```

```
 * filter program.
 */
static int
logfile_write (repository, filter, message, logfp, changes)
    char *repository;
    char *filter;
    char *message;
    FILE *logfp;
    List *changes;
{
    FILE *pipefp;
    char *prog;
    char *cp;
    int c;
    int pipestatus;
    char *fmt_percent;    /* the location of the percent sign
                             that starts the format string. */

    /* The user may specify a format string as part of the filter.
       Originally, '%s' was the only valid string.  The string that
       was substituted for it was:

         <repository-name> <file1> <file2> <file3> ...

       Each file was either a new directory/import (T_TITLE), or a
       added (T_ADDED), modified (T_MODIFIED), or removed (T_REMOVED)
       file.

       It is desirable to preserve that behavior so lots of commitlog
       scripts won't die when they get this new code.  At the same
       time, we'd like to pass other information about the files (like
       version numbers, statuses, or checkin times).

       The solution is to allow a format string that allows us to
       specify those other pieces of information.  The format string
       will be composed of '%' followed by a single format character,
       or followed by a set of format characters surrounded by '{' and
       '}' as separators.  The format characters are:

         s = file name
         V = old version number (pre-checkin)
         v = new version number (post-checkin)

       For example, valid format strings are:

         %{}
         %s
         %{s}
         %{sVv}

       There's no reason that more items couldn't be added (like
       modification date or file status [added, modified, updated,
       etc.]) - the code modifications would be minimal (logmsg.c
       (title_proc) and commit.c (check_fileproc)).

       The output will be a string of tokens separated by spaces.  For
       backwards compatibility, the the first token will be the
       repository name.  The rest of the tokens will be
       comma-delimited lists of the information requested in the
       format string.  For example, if '/u/src/master' is the
       repository, '%{sVv}' is the format string, and three files
       (ChangeLog, Makefile, foo.c) were modified, the output might
       be:

         /u/src/master ChangeLog,1.1,1.2 Makefile,1.3,1.4 foo.c,1.12,1.13

       Why this duplicates the old behavior when the format string is
       '%s' is left as an exercise for the reader. */

    fmt_percent = strchr (filter, '%');
    if (fmt_percent)
    {
        int len;
        char *srepos;
        char *fmt_begin, *fmt_end; /* beginning and end of the
                                      format string specified in
                                      filter. */
        char *fmt_continue;       /* where the string continues
                                      after the format string (we
                                      might skip a '}') somewhere
                                      in there... */

        /* Grab the format string. */

        if ((*(fmt_percent + 1) == ' ') || (*(fmt_percent + 1) == '\0'))
        {
            /* The percent stands alone.  This is an error.  We could
               be treating ' ' like any other formatting character, but
               using it as a formatting character seems like it would be
               a mistake.  */
```

```
720             /* Would be nice to also be giving the line number.  */
                error (0, 0, "loginfo: '%%' not followed by formatting character");
                fmt_begin = fmt_percent + 1;
                fmt_end = fmt_begin;
                fmt_continue = fmt_begin;
            }
            else if (*(fmt_percent + 1) == '{')
            {
                /* The percent has a set of characters following it. */

730             fmt_begin = fmt_percent + 2;
                fmt_end = strchr (fmt_begin, '}');
                if (fmt_end)
                {
                    /* Skip over the '}' character. */

                    fmt_continue = fmt_end + 1;
                }
                else
                {
740                 /* There was no close brace - assume that format
                       string continues to the end of the line. */

                    /* Would be nice to also be giving the line number.  */
                    error (0, 0, "loginfo: '}' missing");
                    fmt_end = fmt_begin + strlen (fmt_begin);
                    fmt_continue = fmt_end;
                }
            }
            else
750         {
                /* The percent has a single character following it.  FIXME:
                   \%\% should expand to a regular percent sign.  */

                fmt_begin = fmt_percent + 1;
                fmt_end = fmt_begin + 1;
                fmt_continue = fmt_end;
            }

            len = fmt_end - fmt_begin;
760         str_list_format = xmalloc (sizeof (char) * (len + 1));
            strncpy (str_list_format, fmt_begin, len);
            str_list_format[len] = '\0';

            /* Allocate an initial chunk of memory.  As we build up the string
               we will realloc it.  */
            if (!str_list)
                str_list = xmalloc (1);
            str_list[0] = '\0';

770         /* Add entries to the string.  Don't bother looking for
               entries if the format string is empty. */

            if (str_list_format[0] != '\0')
            {
                type = T_TITLE;
                (void) walklist (changes, title_proc, NULL);
                type = T_ADDED;
                (void) walklist (changes, title_proc, NULL);
                type = T_MODIFIED;
780             (void) walklist (changes, title_proc, NULL);
                type = T_REMOVED;
                (void) walklist (changes, title_proc, NULL);
            }

            free (str_list_format);

            /* Construct the final string. */

            srepos = Short_Repository (repository);
790
            prog = xmalloc ((fmt_percent - filter) + strlen (srepos)
                            + strlen (str_list) + strlen (fmt_continue)
                            + 10);
            (void) strncpy (prog, filter, fmt_percent - filter);
            prog[fmt_percent - filter] = '\0';
            (void) strcat (prog, "'");
            (void) strcat (prog, srepos);
            (void) strcat (prog, str_list);
            (void) strcat (prog, "'");
800         (void) strcat (prog, fmt_continue);

            /* To be nice, free up some memory. */

            free (str_list);
            str_list = (char *) NULL;
        }
        else
        {
```

```
            /* There's no format string. */
            prog = xstrdup (filter);
        }

        if ((pipefp = run_popen (prog, "w")) == NULL)
        {
            if (!noexec)
                error (0, 0, "cannot write entry to log filter: %s", prog);
            free (prog);
            return (1);
        }
        (void) fprintf (pipefp, "Update of %s\n", repository);
        (void) fprintf (pipefp, "In directory %s:", hostname);
        cp = xgetwd ();
        if (cp == NULL)
            fprintf (pipefp, "<cannot get working directory: %s>\n\n",
                     strerror (errno));
        else
        {
            fprintf (pipefp, "%s\n\n", cp);
            free (cp);
        }

        setup_tmpfile (pipefp, "", changes);
        (void) fprintf (pipefp, "Log Message:\n%s\n", message);
        if (logfp != (FILE *) 0)
        {
            (void) fprintf (pipefp, "Status:\n");
            rewind (logfp);
            while ((c = getc (logfp)) != EOF)
                (void) putc ((char) c, pipefp);
        }
        free (prog);
        pipestatus = pclose (pipefp);
        return ((pipestatus == -1) || (pipestatus == 127)) ? 1 : 0;
    }

    /*
     * We choose to use the *last* match within the editinfo file for this
     * repository.  This allows us to have a global editinfo program for the
     * root of some hierarchy, for example, and different ones within different
     * sub-directories of the root (like a special checker for changes made to
     * the "src" directory versus changes made to the "doc" or "test"
     * directories.
     */
    /* ARGSUSED */
    static int
    editinfo_proc(repository, editor)
        char *repository;
        char *editor;
    {
        /* nothing to do if the last match is the same as this one */
        if (editinfo_editor && strcmp (editinfo_editor, editor) == 0)
            return (0);
        if (editinfo_editor)
            free (editinfo_editor);

        editinfo_editor = xstrdup (editor);
        return (0);
    }

    /*  This routine is calld by Parse_Info.  it asigns the name of the
     *  message verification script to the global variable verify_script
     */
    static int
    verifymsg_proc (repository, script)
        char *repository;
        char *script;
    {
        if (verifymsg_script && strcmp (verifymsg_script, script) == 0)
            return (0);
        if (verifymsg_script)
            free (verifymsg_script);
        verifymsg_script = xstrdup (script);
        return (0);
    }
```

A.36 main.c

```
/*
 *    Copyright (c) 1992, Brian Berliner and Jeff Polk
 *    Copyright (c) 1989-1992, Brian Berliner
 *
 *    You may distribute under the terms of the GNU General Public License
 *    as specified in the README file that comes with the CVS source distribution.
 *
 * This is the main C driver for the CVS system.
 *
 * Credit to Dick Grune, Vrije Universiteit, Amsterdam, for writing
 * the shell-script CVS system that this is based on.
 *
 */

#include "cvs.h"

#ifdef HAVE_WINSOCK_H
#include <winsock.h>
#else
extern int gethostname ();
#endif

char *program_name;
char *program_path;
char *command_name;

/* I'd dynamically allocate this, but it seems like gethostname
   requires a fixed size array.  If I'm remembering the RFCs right,
   256 should be enough.  */
#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN  256
#endif

char hostname[MAXHOSTNAMELEN];

int use_editor = 1;
int use_cvsrc = 1;
int cvswrite = !CVSREAD_DFLT;
int really_quiet = 0;
int quiet = 0;
int trace = 0;
int noexec = 0;
int logoff = 0;

/* Set if we should be writing CVSADM directories at top level.  At
   least for now we'll make the default be off (the CVS 1.9, not CVS
   1.9.2, behavior). */
int top_level_admin = 0;

mode_t cvsumask = UMASK_DFLT;

char *CurDir;

/*
 * Defaults, for the environment variables that are not set
 */
char *Tmpdir = TMPDIR_DFLT;
char *Editor = EDITOR_DFLT;

static const struct cmd
{
    char *fullname;        /* Full name of the function (e.g. "commit") */

    /* Synonyms for the command, nick1 and nick2.  We supply them
       mostly for two reasons: (1) CVS has always supported them, and
       we need to maintain compatibility, (2) if there is a need for a
       version which is shorter than the fullname, for ease in typing.
       Synonyms have the disadvantage that people will see "new" and
       then have to think about it, or look it up, to realize that is
       the operation they know as "add".  Also, this means that one
       cannot create a command "cvs new" with a different meaning.  So
       new synonyms are probably best used sparingly, and where used
       should be abbreviations of the fullname (preferably consisting
       of the first 2 or 3 or so letters).

       One thing that some systems do is to recognize any unique
       abbreviation, for example "annotat" "annota", etc., for
       "annotate".  The problem with this is that scripts and user
       habits will expect a certain abbreviation to be unique, and in
       a future release of CVS it may not be.  So it is better to
       accept only an explicit list of abbreviations and plan on
       supporting them in the future as well as now.  */

    char *nick1;
    char *nick2;

    int (*func) ();          /* Function takes (argc, argv) arguments. */
} cmds[] =
```

```
{
    { "add",      "ad",      "new",       add },
    { "admin",    "adm",     "rcs",       admin },
    { "annotate", "ann",     NULL,        annotate },
    { "checkout", "co",      "get",       checkout },
    { "commit",   "ci",      "com",       commit },
    { "diff",     "di",      "dif",       diff },
    { "edit",     NULL,      NULL,        edit },
    { "editors",  NULL,      NULL,        editors },
    { "export",   "exp",     "ex",        checkout },
    { "history",  "hi",      "his",       history },
    { "import",   "im",      "imp",       import },
    { "init",     NULL,      NULL,        init },
#ifdef SERVER_SUPPORT
    { "kserver",  NULL,      NULL,        server }, /* placeholder */
#endif
    { "log",      "lo",      "rlog",      cvslog },
#ifdef AUTH_CLIENT_SUPPORT
    { "login",    "logon",   "lgn",       login },
    { "logout",   NULL,      NULL,        logout },
#ifdef SERVER_SUPPORT
    { "pserver",  NULL,      NULL,        server }, /* placeholder */
#endif
#endif /* AUTH_CLIENT_SUPPORT */
    { "rdiff",    "patch",   "pa",        patch },
    { "release",  "re",      "rel",       release },
    { "remove",   "rm",      "delete",    cvsremove },
    { "status",   "st",      "stat",      cvsstatus },
    { "rtag",     "rt",      "rfreeze",   rtag },
    { "tag",      "ta",      "freeze",    cvstag },
    { "unedit",   NULL,      NULL,        unedit },
    { "update",   "up",      "upd",       update },
    { "watch",    NULL,      NULL,        watch },
    { "watchers", NULL,      NULL,        watchers },
#ifdef SERVER_SUPPORT
    { "server",   NULL,      NULL,        server },
#endif
    { NULL, NULL, NULL, NULL },
};

static const char *const usg[] =
{
    /* CVS usage messages never have followed the GNU convention of
       putting metavariables in uppercase.  I don't know whether that
       is a good convention or not, but if it changes it would have to
       change in all the usage messages.  For now, they consistently
       use lowercase, as far as I know.  Puncutation is pretty funky,
       though.  Sometimes they use none, as here.  Sometimes they use
       single quotes (not the TeX-ish `' stuff), as in --help-options.
       Sometimes they use double quotes, as in cvs -H add.

       Most (not all) of the usage messages seem to have periods at
       the end of each line.  I haven't tried to duplicate this style
       in --help as it is a rather different format from the rest.  */

    "Usage: %s [cvs-options] command [command-options-and-arguments]\n",
    "  where cvs-options are -q, -n, etc.\n",
    "    (specify --help-options for a list of options)\n",
    "  where command is add, admin, etc.\n",
    "    (specify --help-commands for a list of commands\n",
    "     or --help-synonyms for a list of command synonyms)\n",
    "  where command-options-and-arguments depend on the specific command\n",
    "    (specify -H followed by a command name for command-specific help)\n",
    "  Specify --help to receive this message\n",
    "\n",

    /* Some people think that a bug-reporting address should go here.  IMHO,
       the web sites are better because anything else is very likely to go
       obsolete in the years between a release and when someone might be
       reading this help.  Besides, we could never adequately discuss
       bug reporting in a concise enough way to put in a help message.  */

    /* I was going to put this at the top, but usage() wants the %s to
       be in the first line.  */
    "The Concurrent Versions System (CVS) is a tool for version control.\n",
    /* I really don't think I want to try to define "version control"
       in one line.  I'm not sure one can get more concise than the
       paragraph in ../cvs.spec without assuming the reader knows what
       version control means.  */

    "For CVS updates and additional information, see\n",
    "    Cyclic Software at http://www.cyclic.com/ or\n",
    "    Pascal Molli's CVS site at http://www.loria.fr/~molli/cvs-index.html\n",
    NULL,
};

static const char *const cmd_usage[] =
{
    "CVS commands are:\n",
```

```
    "       add         Add a new file/directory to the repository\n",
    "       admin       Administration front end for rcs\n",
    "       annotate    Show last revision where each line was modified\n",
    "       checkout    Checkout sources for editing\n",
    "       commit      Check files into the repository\n",
    "       diff        Show differences between revisions\n",
    "       edit        Get ready to edit a watched file\n",
    "       editors     See who is editing a watched file\n",
    "       export      Export sources from CVS, similar to checkout\n",
    "       history     Show repository access history\n",
    "       import      Import sources into CVS, using vendor branches\n",
    "       init        Create a CVS repository if it doesn't exist\n",
    "       log         Print out history information for files\n",
#ifdef AUTH_CLIENT_SUPPORT
    "       login       Prompt for password for authenticating server.\n",
    "       logout      Removes entry in .cvspass for remote repository.\n",
#endif /* AUTH_CLIENT_SUPPORT */
    "       rdiff       Create 'patch' format diffs between releases\n",
    "       release     Indicate that a Module is no longer in use\n",
    "       remove      Remove an entry from the repository\n",
    "       rtag        Add a symbolic tag to a module\n",
    "       status      Display status information on checked out files\n",
    "       tag         Add a symbolic tag to checked out version of files\n",
    "       unedit      Undo an edit command\n",
    "       update      Bring work tree in sync with repository\n",
    "       watch       Set watches\n",
    "       watchers    See who is watching a file\n",
    "(Specify the --help option for a list of other help options)\n",
    NULL,
};

static const char *const opt_usage[] =
{
    "CVS global options (specified before the command name) are:\n",
    "    -H         Displays usage information for command.\n",
    "    -Q         Cause CVS to be really quiet.\n",
    "    -q         Cause CVS to be somewhat quiet.\n",
    "    -r         Make checked-out files read-only.\n",
    "    -w         Make checked-out files read-write (default).\n",
    "    -l         Turn history logging off.\n",
    "    -n         Do not execute anything that will change the disk.\n",
    "    -t         Show trace of program execution -- try with -n.\n",
    "    -v         CVS version and copyright.\n",
    "    -b bindir   Find RCS programs in 'bindir'.\n",
    "    -T tmpdir   Use 'tmpdir' for temporary files.\n",
    "    -e editor   Use 'editor' for editing log information.\n",
    "    -d CVS_root  Overrides $CVSROOT as the root of the CVS tree.\n",
    "    -f         Do not use the ~/.cvsrc file.\n",
#ifdef CLIENT_SUPPORT
    "    -z #       Use compression level '#' for net traffic.\n",
#ifdef ENCRYPTION
    "    -x         Encrypt all net traffic.\n",
#endif
    "    -a         Authenticate all net traffic.\n",
#endif
    "    -s VAR=VAL   Set CVS user variable.\n",
    "(Specify the --help option for a list of other help options)\n",
    NULL
};

static const char * const*
cmd_synonyms ()
{
    char ** synonyms;
    char ** line;
    const struct cmd *c = &cmds[0];
    /* Three more for title, "specify -help" line, and NULL.  */
    int numcmds = 3;

    while (c->fullname != NULL)
    {
        numcmds++;
        c++;
    }

    synonyms = (char **) xmalloc(numcmds * sizeof(char *));
    line = synonyms;
    *line++ = "CVS command synonyms are:\n";
    for (c = &cmds[0]; c->fullname != NULL; c++)
    {
        if (c->nick1 || c->nick2)
        {
            *line = xmalloc (strlen (c->fullname)
                             + (c->nick1 != NULL ? strlen (c->nick1) : 0)
                             + (c->nick2 != NULL ? strlen (c->nick2) : 0)
                             + 40);
            sprintf(*line, "       %-12s %s %s\n", c->fullname,
                    c->nick1 ? c->nick1 : "",
                    c->nick2 ? c->nick2 : "");
            line++;
```

```
        }
    }
    *line++ = "(Specify the --help option for a list of other help options)\n";
    *line = NULL;

    return (const char * const*) synonyms; /* will never be freed */
}


unsigned long int
lookup_command_attribute (cmd_name)
     char *cmd_name;
{
    unsigned long int ret = 0;

    if (strcmp (cmd_name, "import") != 0)
    {
        ret |= CVS_CMD_IGNORE_ADMROOT;
    }


    /* The following commands do not use a checked-out working
       directory.  We conservatively assume that everything else does.
       Feel free to add to this list if you are _certain_ something
       something doesn't use the WD. */
    if ((strcmp (cmd_name, "checkout") != 0) &&
        (strcmp (cmd_name, "init") != 0) &&
        (strcmp (cmd_name, "login") != 0) &&
        (strcmp (cmd_name, "logout") != 0) &&
        (strcmp (cmd_name, "rdiff") != 0) &&
        (strcmp (cmd_name, "release") != 0) &&
        (strcmp (cmd_name, "rtag") != 0))
    {
        ret |= CVS_CMD_USES_WORK_DIR;
    }


    /* The following commands do not modify the repository; we
       conservatively assume that everything else does.  Feel free to
       add to this list if you are _certain_ something is safe. */
    if ((strcmp (cmd_name, "annotate") != 0) &&
        (strcmp (cmd_name, "checkout") != 0) &&
        (strcmp (cmd_name, "diff") != 0) &&
        (strcmp (cmd_name, "rdiff") != 0) &&
        (strcmp (cmd_name, "update") != 0) &&
        (strcmp (cmd_name, "history") != 0) &&
        (strcmp (cmd_name, "editors") != 0) &&
        (strcmp (cmd_name, "export") != 0) &&
        (strcmp (cmd_name, "history") != 0) &&
        (strcmp (cmd_name, "log") != 0) &&
        (strcmp (cmd_name, "noop") != 0) &&
        (strcmp (cmd_name, "watchers") != 0) &&
        (strcmp (cmd_name, "status") != 0))
    {
        ret |= CVS_CMD_MODIFIES_REPOSITORY;
    }

    return ret;
}


static RETSIGTYPE
main_cleanup (sig)
    int sig;
{
#ifndef DONT_USE_SIGNALS
    const char *name;
    char temp[10];

    switch (sig)
    {
#ifdef SIGHUP
    case SIGHUP:
        name = "hangup";
        break;
#endif
#ifdef SIGINT
    case SIGINT:
        name = "interrupt";
        break;
#endif
#ifdef SIGQUIT
    case SIGQUIT:
        name = "quit";
        break;
#endif
#ifdef SIGPIPE
    case SIGPIPE:
        name = "broken pipe";
        break;
```

```
#endif
#ifdef SIGTERM
    case SIGTERM:
        name = "termination";
        break;
#endif
    default:
        /* This case should never be reached, because we list above all
           the signals for which we actually establish a signal handler.  */
        sprintf (temp, "%d", sig);
        name = temp;
        break;
    }

    error (1, 0, "received %s signal", name);
#endif /* !DONT_USE_SIGNALS */
}

int
main (argc, argv)
    int argc;
    char **argv;
{
    char *CVSroot = CVSROOT_DFLT;
    extern char *version_string;
    extern char *config_string;
    char *cp, *end;
    const struct cmd *cm;
    int c, err = 0;
    int tmpdir_update_env, cvs_update_env;
    int free_CVSroot = 0;
    int free_Editor = 0;
    int free_Tmpdir = 0;

    int help = 0;           /* Has the user asked for help?  This
                               lets us support the 'cvs -H cmd'
                               convention to give help for cmd. */
    static struct option long_options[] =
    {
        {"help", 0, NULL, 'H'},
        {"version", 0, NULL, 'v'},
        {"help-commands", 0, NULL, 1},
        {"help-synonyms", 0, NULL, 2},
        {"help-options", 0, NULL, 4},
        {"allow-root", required_argument, NULL, 3},
        {0, 0, 0, 0}
    };
    /* 'getopt_long' stores the option index here, but right now we
       don't use it. */
    int option_index = 0;
    int need_to_create_root = 0;

#ifdef SYSTEM_INITIALIZE
    /* Hook for OS-specific behavior, for example socket subsystems on
       NT and OS2 or dealing with windows and arguments on Mac.  */
    SYSTEM_INITIALIZE (&argc, &argv);
#endif

#ifdef HAVE_TZSET
    /* On systems that have tzset (which is almost all the ones I know
       of), it's a good idea to call it.  */
    tzset ();
#endif

    /*
     * Just save the last component of the path for error messages
     */
    program_path = xstrdup (argv[0]);
#ifdef ARGV0_NOT_PROGRAM_NAME
    /* On some systems, e.g. VMS, argv[0] is not the name of the command
       which the user types to invoke the program.  */
    program_name = "cvs";
#else
    program_name = last_component (argv[0]);
#endif

    /*
     * Query the environment variables up-front, so that
     * they can be overridden by command line arguments
     */
    cvs_update_env = 0;
    tmpdir_update_env = *Tmpdir; /* TMPDIR_DFLT must be set */
    if ((cp = getenv (TMPDIR_ENV)) != NULL)
    {
        Tmpdir = cp;
        tmpdir_update_env = 0;  /* it's already there */
    }
    if ((cp = getenv (EDITOR1_ENV)) != NULL)
        Editor = cp;
    else if ((cp = getenv (EDITOR2_ENV)) != NULL)
```

```
        Editor = cp;
    else if ((cp = getenv (EDITOR3_ENV)) != NULL)
        Editor = cp;
    if ((cp = getenv (CVSROOT_ENV)) != NULL)
    {
        CVSroot = cp;
        cvs_update_env = 0;       /* it's already there */
    }
    if (getenv (CVSREAD_ENV) != NULL)
        cvswrite = 0;

    /* Set this to 0 to force getopt initialization.  getopt() sets
       this to 1 internally.  */
    optind = 0;

    /* We have to parse the options twice because else there is no
       chance to avoid reading the global options from ".cvsrc".  Set
       opterr to 0 for avoiding error messages about invalid options.
       */
    opterr = 0;

    while ((c = getopt_long
            (argc, argv, "+f", NULL, NULL))
           != EOF)
    {
        if (c == 'f')
            use_cvsrc = 0;
    }

    /*
     * Scan cvsrc file for global options.
     */
    if (use_cvsrc)
        read_cvsrc (&argc, &argv, "cvs");

    optind = 0;
    opterr = 1;

    while ((c = getopt_long
            (argc, argv, "+Qqrwtnlvb:T:e:d:Hfz:s:xa", long_options, &option_index))
           != EOF)
    {
        switch (c)
        {
            case 1:
                /* --help-commands */
                usage (cmd_usage);
                break;
            case 2:
                /* --help-synonyms */
                usage (cmd_synonyms());
                break;
            case 4:
                /* --help-options */
                usage (opt_usage);
                break;
            case 3:
                /* --allow-root */
                root_allow_add (optarg);
                break;
            case 'Q':
                really_quiet = 1;
                /* FALL THROUGH */
            case 'q':
                quiet = 1;
                break;
            case 'r':
                cvswrite = 0;
                break;
            case 'w':
                cvswrite = 1;
                break;
            case 't':
                trace = 1;
                break;
            case 'n':
                noexec = 1;
            case 'l':                 /* Fall through */
                logoff = 1;
                break;
            case 'v':
                /* Having the year here is a good idea, so people have
                   some idea of how long ago their version of CVS was
                   released.  */
                (void) fputs (version_string, stdout);
                (void) fputs (config_string, stdout);
                (void) fputs ("\n", stdout);
                (void) fputs ("\
Copyright (c) 1989-1998 Brian Berliner, david d 'zoo' zuhn, \n\
                      Jeff Polk, and other authors\n", stdout);
```

```
            (void) fputs ("\n", stdout);
            (void) fputs ("CVS may be copied only under the terms of the GNU General Public License,\n", stdout);
            (void) fputs ("a copy of which can be found with the CVS distribution kit.\n", stdout);
            (void) fputs ("\n", stdout);

            (void) fputs ("Specify the --help option for further information about CVS\n", stdout);

            exit (0);
            break;
          case 'b':
            /* This option used to specify the directory for RCS
               executables.  But since we don't run them any more,
               this is a noop.  Silently ignore it so that .cvsrc
               and scripts and inetd.conf and such can work with
               either new or old CVS.  */
            break;
          case 'T':
            Tmpdir = xstrdup (optarg);
            free_Tmpdir = 1;
            tmpdir_update_env = 1; /* need to update environment */
            break;
          case 'e':
            Editor = xstrdup (optarg);
            free_Editor = 1;
            break;
          case 'd':
            CVSroot = xstrdup (optarg);
            free_CVSroot = 1;
            cvs_update_env = 1; /* need to update environment */
            break;
          case 'H':
            help = 1;
            break;
          case 'f':
            use_cvsrc = 0; /* unnecessary, since we've done it above */
            break;
          case 'z':
#ifdef CLIENT_SUPPORT
            gzip_level = atoi (optarg);
            if (gzip_level <= 0 || gzip_level > 9)
              error (1, 0,
                     "gzip compression level must be between 1 and 9");
#endif
            /* If no CLIENT_SUPPORT, we just silently ignore the gzip
               level, so that users can have it in their .cvsrc and not
               cause any trouble.  */
            break;
          case 's':
            variable_set (optarg);
            break;
          case 'x':
#ifdef CLIENT_SUPPORT
            cvsencrypt = 1;
#endif /* CLIENT_SUPPORT */
            /* If no CLIENT_SUPPORT, ignore -x, so that users can
               have it in their .cvsrc and not cause any trouble.
               If no ENCRYPTION, we still accept -x, but issue an
               error if we are being run as a client.  */
            break;
          case 'a':
#ifdef CLIENT_SUPPORT
            cvsauthenticate = 1;
#endif
            /* If no CLIENT_SUPPORT, ignore -a, so that users can
               have it in their .cvsrc and not cause any trouble.
               We will issue an error later if stream
               authentication is not supported.  */
            break;
          case '?':
          default:
            usage (usg);
        }
    }

    argc -= optind;
    argv += optind;
    if (argc < 1)
        usage (usg);


    /* Look up the command name.  */

    command_name = argv[0];
    for (cm = cmds; cm->fullname; cm++)
    {
        if (cm->nick1 && !strcmp (command_name, cm->nick1))
            break;
        if (cm->nick2 && !strcmp (command_name, cm->nick2))
            break;
        if (!strcmp (command_name, cm->fullname))
```

```
            break;
    }

    if (!cm->fullname)
        usage (cmd_usage);          /* no match */
    else
        command_name = cm->fullname; /* Global pointer for later use */

    /* This should probably remain a warning, rather than an error,
       for quite a while.  For one thing the version of VC distributed
       with GNU emacs 19.34 invokes 'cvs rlog' instead of 'cvs log'.  */
    if (strcmp (argv[0], "rlog") == 0)
    {
        error (0, 0, "warning: the rlog command is deprecated");
        error (0, 0, "use the synonymous log command instead");
    }

    if (help)
        argc = -1;              /* some functions only check for this */
    else
    {
        /* The user didn't ask for help, so go ahead and authenticate,
           set up CVSROOT, and the rest of it. */

        /* The UMASK environment variable isn't handled with the
           others above, since we don't want to signal errors if the
           user has asked for help.  This won't work if somebody adds
           a command-line flag to set the umask, since we'll have to
           parse it before we get here. */

        if ((cp = getenv (CVSUMASK_ENV)) != NULL)
        {
            /* FIXME: Should be accepting symbolic as well as numeric mask.  */
            cvsumask = strtol (cp, &end, 8) & 0777;
            if (*end != '\0')
                error (1, errno, "invalid umask value in %s (%s)",
                       CVSUMASK_ENV, cp);
        }

#if (defined(AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI)) && defined(SERVER_SUPPORT) \
|| defined (HAVE_KERBEROS) && defined (SERVER_SUPPORT)
        if ((strcmp (command_name, "pserver") == 0) ||
            (strcmp (command_name, "kserver") == 0) ||
            (strcmp (command_name, "gserver") == 0)) {

          static int authenticated = 0;
            /* The reason that -allow-root is not a command option
               is mainly the comment in server() about how argc,argv
               might be from .cvsrc.  I'm not sure about that, and
               I'm not sure it is only true of command options, but
               it seems easier to make it a global option.  */

          if (!authenticated) {
            /* send the list of allowed auth modes to the client */
#ifdef AUTH_SERVER_SUPPORT
            printf ("PASSWORD ");
#endif
#ifdef AUTH_SERVER_SUPPORT
            printf ("GSSAPI ");
#endif
#ifdef AUTH_SERVER_SUPPORT
            printf ("KERBEROS_V4 ");
#endif
            printf ("\n");
            fflush (stdout);

            /* Gets username and password from client, authenticates, then
               switches to run as that user and sends an ACK back to the
               client. */
            server_authenticate_connection ();
            authenticated = 1;
          }

            /* Pretend we were invoked as a plain server.  */
            command_name = "server";

        }

#endif /* (AUTH_SERVER_SUPPORT || HAVE_GSSAPI) && SERVER_SUPPORT */

#ifdef SERVER_SUPPORT
        server_active = strcmp (command_name, "server") == 0;

        /* Fiddling with CVSROOT doesn't make sense if we're running
           in server mode, since the client will send the repository
           directory after the connection is made. */

        if (!server_active)
#endif
        {
```

```
    char *CVSADM_Root;

    /* See if we are able to find a 'better' value for CVSroot
       in the CVSADM_ROOT directory. */

    CVSADM_Root = NULL;

    /* "cvs import" shouldn't check CVS/Root; in general it
       ignores CVS directories and CVS/Root is likely to
       specify a different repository than the one we are
       importing to.  */

    if (lookup_command_attribute (command_name)
        & CVS_CMD_IGNORE_ADMROOT)
    {
        CVSADM_Root = Name_Root((char *) NULL, (char *) NULL);
    }

    if (CVSADM_Root != NULL)
    {
        if (CVSroot == NULL || !cvs_update_env)
        {
            CVSroot = CVSADM_Root;
            cvs_update_env = 1; /* need to update environment */
        }
        /* Let -d override CVS/Root file.  The user might want
           to change the access method, use a different server
           (if there are two server machines which share the
           repository using a networked file system), etc.  */
        else if (
#ifdef CLIENT_SUPPORT
                 !getenv ("CVS_IGNORE_REMOTE_ROOT") &&
#endif
                 strcmp (CVSroot, CVSADM_Root) != 0)
        {
            /* Once we have verified that this root is usable,
               we will want to write it into CVS/Root.

               Don't do it for the "login" command, however.
               Consider: if the user executes "cvs login" with
               the working directory inside an already checked
               out module, we'd incorrectly change the
               CVS/Root file to reflect the CVSROOT of the
               "cvs login" command.  Ahh, the things one
               discovers. */

            if (lookup_command_attribute (command_name)
                & CVS_CMD_USES_WORK_DIR)
            {
                need_to_create_root = 1;
            }

        }
    }

    /* Now we've reconciled CVSROOT from the command line, the
       CVS/Root file, and the environment variable.  Do the
       last sanity checks on the variable. */

    if (! CVSroot)
    {
        error (0, 0,
               "No CVSROOT specified!  Please use the `-d' option");
        error (1, 0,
               "or set the %s environment variable.", CVSROOT_ENV);
    }

    if (! *CVSroot)
    {
        error (0, 0,
               "CVSROOT is set but empty!  Make sure that the");
        error (0, 0,
               "specification of CVSROOT is legal, either via the");
        error (0, 0,
               "`-d' option, the %s environment variable, or the",
               CVSROOT_ENV);
        error (1, 0,
               "CVS/Root file (if any).");
    }

    /* Now we're 100% sure that we have a valid CVSROOT
       variable.  Parse it to see if we're supposed to do
       remote accesses or use a special access method. */

    if (parse_cvsroot (CVSroot))
        error (1, 0, "Bad CVSROOT.");

    /*
     * Check to see if we can write into the history file.  If not,
     * we assume that we can't work in the repository.
```

```
             * BUT, only if the history file exists.
             */

            if (!client_active)
            {
                char *path;
                int save_errno;

                path = xmalloc (strlen (CVSroot_directory)
                                + sizeof (CVSROOTADM)
                                + 20
                                + sizeof (CVSROOTADM_HISTORY));
                (void) sprintf (path, "%s/%s", CVSroot_directory, CVSROOTADM);
                if (!isaccessible (path, R_OK | X_OK))
                {
                    save_errno = errno;
                    /* If this is "cvs init", the root need not exist yet.  */
                    if (strcmp (command_name, "init") != 0)
                    {
                        error (1, save_errno, "%s", path);
                    }
                }
                (void) strcat (path, "/");
                (void) strcat (path, CVSROOTADM_HISTORY);
                if (isfile (path) && !isaccessible (path, R_OK | W_OK))
                {
                    save_errno = errno;
                    error (0, 0, "Sorry, you don't have read/write access to the history file");
                    error (1, save_errno, "%s", path);
                }
                free (path);
            }

#ifdef HAVE_PUTENV
            /* Update the CVSROOT environment variable if necessary. */

            if (cvs_update_env)
            {
                char *env;
                env = xmalloc (strlen (CVSROOT_ENV) + strlen (CVSroot)
                               + 1 + 1);
                (void) sprintf (env, "%s=%s", CVSROOT_ENV, CVSroot);
                (void) putenv (env);
                /* do not free env, as putenv has control of it */
            }
#endif
        }

        /* This is only used for writing into the history file.  For
           remote connections, it might be nice to have hostname
           and/or remote path, on the other hand I'm not sure whether
           it is worth the trouble.  */

#ifdef SERVER_SUPPORT
        if (server_active)
            CurDir = xstrdup ("<remote>");
        else
#endif
        {
            CurDir = xgetwd ();
            if (CurDir == NULL)
                error (1, errno, "cannot get working directory");
        }

        if (Tmpdir == NULL || Tmpdir[0] == '\0')
            Tmpdir = "/tmp";

#ifdef HAVE_PUTENV
        if (tmpdir_update_env)
        {
            char *env;
            env = xmalloc (strlen (TMPDIR_ENV) + strlen (Tmpdir) + 1 + 1);
            (void) sprintf (env, "%s=%s", TMPDIR_ENV, Tmpdir);
            (void) putenv (env);
            /* do not free env, as putenv has control of it */
        }
#endif

#ifndef DONT_USE_SIGNALS
        /* make sure we clean up on error */
#ifdef SIGHUP
        (void) SIG_register (SIGHUP, main_cleanup);
        (void) SIG_register (SIGHUP, Lock_Cleanup);
#endif
#ifdef SIGINT
        (void) SIG_register (SIGINT, main_cleanup);
        (void) SIG_register (SIGINT, Lock_Cleanup);
#endif
#ifdef SIGQUIT
        (void) SIG_register (SIGQUIT, main_cleanup);
```

```
        (void) SIG_register (SIGQUIT, Lock_Cleanup);
#endif
#ifdef SIGPIPE
        (void) SIG_register (SIGPIPE, main_cleanup);
        (void) SIG_register (SIGPIPE, Lock_Cleanup);
#endif
#ifdef SIGTERM
        (void) SIG_register (SIGTERM, main_cleanup);
        (void) SIG_register (SIGTERM, Lock_Cleanup);
#endif
#endif /* !DONT_USE_SIGNALS */

        gethostname(hostname, sizeof (hostname));

#ifdef KLUDGE_FOR_WNT_TESTSUITE
        /* Probably the need for this will go away at some point once
           we call fflush enough places (e.g. fflush (stdout) in
           cvs_outerr).  */
        (void) setvbuf (stdout, (char *) NULL, _IONBF, 0);
        (void) setvbuf (stderr, (char *) NULL, _IONBF, 0);
#endif /* KLUDGE_FOR_WNT_TESTSUITE */

        if (use_cvsrc)
            read_cvsrc (&argc, &argv, command_name);

        /* Parse the CVSROOT/config file, but only for local.  For the
           server, we parse it after we know $CVSROOT.  For the
           client, it doesn't get parsed at all, obviously.  The
           presence of the parse_config call here is not mean to
           predetermine whether CVSROOT/config overrides things from
           read_cvsrc and other such places or vice versa.  That sort
           of thing probably needs more thought.  */
        if (1
#ifdef SERVER_SUPPORT
            && !server_active
#endif
#ifdef CLIENT_SUPPORT
            && !client_active
#endif
            )
        {
            /* If there was an error parsing the config file, parse_config
               already printed an error.  We keep going.  Why?  Because
               if we didn't, then there would be no way to check in a new
               CVSROOT/config file to fix the broken one!  */
            parse_config (CVSroot_directory);
        }
    } /* end of stuff that gets done if the user DOESN'T ask for help */

    err = (*(cm->func)) (argc, argv);
    client_process_remotes (cm->func, argc, argv);

    if (need_to_create_root)
    {
        /* Update the CVS/Root file.  We might want to do this in
           all directories that we recurse into, but currently we
           don't.  Note that if there is an error writing the file,
           we give an error/warning.  This is so if users try to rewrite
           CVS/Root with the -d option (a documented feature), they will
           either succeed, or be told why it didn't work.  */
        Create_Root (NULL, CVSroot);
    }

    Lock_Cleanup ();

    free (program_path);
    if (free_CVSroot)
        free (CVSroot);
    if (free_Editor)
        free (Editor);
    if (free_Tmpdir)
        free (Tmpdir);
    root_allow_free ();

#ifdef SYSTEM_CLEANUP
    /* Hook for OS-specific behavior, for example socket subsystems on
       NT and OS2 or dealing with windows and arguments on Mac.  */
    SYSTEM_CLEANUP ();
#endif

    /* This is exit rather than return because apparently that keeps
       some tools which check for memory leaks happier.  */
    exit (err ? EXIT_FAILURE : 0);
        /* Keep picky/stupid compilers (e.g. Visual C++ 5.0) happy.  */
        return 0;
}

char *
Make_Date (rawdate)
    char *rawdate;
```

```
{
    time_t unixtime;

    unixtime = get_date (rawdate, (struct timeb *) NULL);
    if (unixtime == (time_t) - 1)
        error (1, 0, "Can't parse date/time: %s", rawdate);
    return date_from_time_t (unixtime);
}

/* Convert a time_t to an RCS format date.  This is mainly for the
   use of "cvs history", because the CVSROOT/history file contains
   time_t format dates; most parts of CVS will want to avoid using
   time_t's directly, and instead use RCS_datecmp, Make_Date, &c.
   Assuming that the time_t is in GMT (as it generally should be),
   then the result will be in GMT too.

   Returns a newly malloc'd string.  */

char *
date_from_time_t (unixtime)
    time_t unixtime;
{
    struct tm *ftm;
    char date[MAXDATELEN];
    char *ret;

    ftm = gmtime (&unixtime);
    if (ftm == NULL)
        /* This is a system, like VMS, where the system clock is in local
           time.  Hopefully using localtime here matches the "zero timezone"
           hack I added to get_date (get_date of course being the relevant
           issue for Make_Date, and for history.c too I think).  */
        ftm = localtime (&unixtime);

    (void) sprintf (date, DATEFORM,
                    ftm->tm_year + (ftm->tm_year < 100 ? 0 : 1900),
                    ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
                    ftm->tm_min, ftm->tm_sec);
    ret = xstrdup (date);
    return (ret);
}

void
usage (cpp)
    register const char *const *cpp;
{
    (void) fprintf (stderr, *cpp++, program_name, command_name);
    for (; *cpp; cpp++)
        (void) fprintf (stderr, *cpp);
    error_exit ();
}
```

A.37 mkmodules.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS kit. */

#include "cvs.h"
#include "savecwd.h"
#include "getline.h"

#ifndef DBLKSIZ
#define DBLKSIZ 4096            /* since GNU ndbm doesn't define it */
#endif

static int checkout_file PROTO((char *file, char *temp));
static char *make_tempfile PROTO((void));
static void rename_rcsfile PROTO((char *temp, char *real));

#ifndef MY_NDBM
static void rename_dbmfile PROTO((char *temp));
static void write_dbmfile PROTO((char *temp));
#endif                          /* !MY_NDBM */

/* Structure which describes an administrative file. */
struct admin_file {
  /* Name of the file, within the CVSROOT directory. */
  char *filename;

  /* This is a one line description of what the file is for.  It is not
     currently used, although one wonders whether it should be, somehow.
     If NULL, then don't process this file in mkmodules (FIXME?: a bit of
     a kludge; probably should replace this with a flags field). */
  char *errormsg;

  /* Contents which the file should have in a new repository.  To avoid
     problems with brain-dead compilers which choke on long string constants,
     this is a pointer to an array of char * terminated by NULL--each of
     the strings is concatenated.

     If this field is NULL, the file is not created in a new
     repository, but it can be added with "cvs add" (just as if one
     had created the repository with a version of CVS which didn't
     know about the file) and the checked-out copy will be updated
     without having to add it to checkoutlist. */
  const char * const *contents;
};

static const char *const loginfo_contents[] = {
   "# The \"loginfo\" file controls where \"cvs commit\" log information\n",
   "# is sent.  The first entry on a line is a regular expression which must match\n",
   "# the directory that the change is being made to, relative to the\n",
   "# $CVSROOT.  If a match is found, then the remainder of the line is a filter\n",
   "# program that should expect log information on its standard input.\n",
   "#\n",
   "# If the repository name does not match any of the regular expressions in this\n",
   "# file, the \"DEFAULT\" line is used, if it is specified.\n",
   "#\n",
   "# If the name ALL appears as a regular expression it is always used\n",
   "# in addition to the first matching regex or DEFAULT.\n",
   "#\n",
   "# You may specify a format string as part of the\n",
   "# filter.  The string is composed of a `%' followed\n",
   "# by a single format character, or followed by a set of format\n",
   "# characters surrounded by `{' and `}' as separators.  The format\n",
   "# characters are:\n",
   "#\n",
   "#   s = file name\n",
   "#   V = old version number (pre-checkin)\n",
   "#   v = new version number (post-checkin)\n",
   "#\n",
   "# For example:\n",
   "#DEFAULT (echo \"\"; id; echo %s; date; cat) >> $CVSROOT/CVSROOT/commitlog\n",
   "# or\n",
   "#DEFAULT (echo \"\"; id; echo %{sVv}; date; cat) >> $CVSROOT/CVSROOT/commitlog\n",
   NULL
};

static const char *const rcsinfo_contents[] = {
   "# The \"rcsinfo\" file is used to control templates with which the editor\n",
   "# is invoked on commit and import.\n",
   "#\n",
   "# The first entry on a line is a regular expression which is tested\n",
   "# against the directory that the change is being made to, relative to the\n",
   "# $CVSROOT.  For the first match that is found, then the remainder of the\n",
   "# line is the name of the file that contains the template.\n",
   "#\n",
   "# If the repository name does not match any of the regular expressions in this\n",
```

```
    "# file, the \"DEFAULT\" line is used, if it is specified.\n",
    "#\n",
    "# If the name \"ALL\" appears as a regular expression it is always used\n",
    "# in addition to the first matching regex or \"DEFAULT\".\n",
    NULL
};

static const char *const editinfo_contents[] = {
    "# The \"editinfo\" file is used to allow verification of logging\n",
    "# information.  It works best when a template (as specified in the\n",
    "# rcsinfo file) is provided for the logging procedure.  Given a\n",
    "# template with locations for, a bug-id number, a list of people who\n",
    "# reviewed the code before it can be checked in, and an external\n",
    "# process to catalog the differences that were code reviewed, the\n",
    "# following test can be applied to the code:\n",
    "#\n",
    "#   Making sure that the entered bug-id number is correct.\n",
    "#   Validating that the code that was reviewed is indeed the code being\n",
    "#       checked in (using the bug-id number or a seperate review\n",
    "#       number to identify this particular code set.).\n",
    "#\n",
    "# If any of the above test failed, then the commit would be aborted.\n",
    "#\n",
    "# Actions such as mailing a copy of the report to each reviewer are\n",
    "# better handled by an entry in the loginfo file.\n",
    "#\n",
    "# One thing that should be noted is the the ALL keyword is not\n",
    "# supported.  There can be only one entry that matches a given\n",
    "# repository.\n",
    NULL
};

static const char *const verifymsg_contents[] = {
    "# The \"verifymsg\" file is used to allow verification of logging\n",
    "# information.  It works best when a template (as specified in the\n",
    "# rcsinfo file) is provided for the logging procedure.  Given a\n",
    "# template with locations for, a bug-id number, a list of people who\n",
    "# reviewed the code before it can be checked in, and an external\n",
    "# process to catalog the differences that were code reviewed, the\n",
    "# following test can be applied to the code:\n",
    "#\n",
    "#   Making sure that the entered bug-id number is correct.\n",
    "#   Validating that the code that was reviewed is indeed the code being\n",
    "#       checked in (using the bug-id number or a seperate review\n",
    "#       number to identify this particular code set.).\n",
    "#\n",
    "# If any of the above test failed, then the commit would be aborted.\n",
    "#\n",
    "# Actions such as mailing a copy of the report to each reviewer are\n",
    "# better handled by an entry in the loginfo file.\n",
    "#\n",
    "# One thing that should be noted is the the ALL keyword is not\n",
    "# supported.  There can be only one entry that matches a given\n",
    "# repository.\n",
    NULL
};

static const char *const commitinfo_contents[] = {
    "# The \"commitinfo\" file is used to control pre-commit checks.\n",
    "# The filter on the right is invoked with the repository and a list \n",
    "# of files to check.  A non-zero exit of the filter program will \n",
    "# cause the commit to be aborted.\n",
    "#\n",
    "# The first entry on a line is a regular expression which is tested\n",
    "# against the directory that the change is being committed to, relative\n",
    "# to the $CVSROOT.  For the first match that is found, then the remainder\n",
    "# of the line is the name of the filter to run.\n",
    "#\n",
    "# If the repository name does not match any of the regular expressions in this\n",
    "# file, the \"DEFAULT\" line is used, if it is specified.\n",
    "#\n",
    "# If the name \"ALL\" appears as a regular expression it is always used\n",
    "# in addition to the first matching regex or \"DEFAULT\".\n",
    NULL
};

static const char *const taginfo_contents[] = {
    "# The \"taginfo\" file is used to control pre-tag checks.\n",
    "# The filter on the right is invoked with the following arguments:\n",
    "#\n",
    "# $1 -- tagname\n",
    "# $2 -- operation \"add\" for tag, \"mov\" for tag -F, and \"del\" for tag -d\n",
    "# $3 -- repository\n",
    "# $4->  file revision [file revision ...]\n",
    "#\n",
    "# A non-zero exit of the filter program will cause the tag to be aborted.\n",
    "#\n",
    "# The first entry on a line is a regular expression which is tested\n",
    "# against the directory that the change is being committed to, relative\n",
    "# to the $CVSROOT.  For the first match that is found, then the remainder\n",
```

```
    "# of the line is the name of the filter to run.\n",
    "#\n",
    "# If the repository name does not match any of the regular expressions in this\n",
    "# file, the \"DEFAULT\" line is used, if it is specified.\n",
    "#\n",
    "# If the name \"ALL\" appears as a regular expression it is always used\n",
    "# in addition to the first matching regex or \"DEFAULT\".\n",
    NULL
};

static const char *const checkoutlist_contents[] = {
    "# The \"checkoutlist\" file is used to support additional version controlled\n",
    "# administrative files in $CVSROOT/CVSROOT, such as template files.\n",
    "#\n",
    "# The first entry on a line is a filename which will be checked out from\n",
    "# the corresponding RCS file in the $CVSROOT/CVSROOT directory.\n",
    "# The remainder of the line is an error message to use if the file cannot\n",
    "# be checked out.\n",
    "#\n",
    "# File format:\n",
    "#\n",
    "#  [<whitespace>]<filename><whitespace><error message><end-of-line>\n",
    "#\n",
    "# comment lines begin with '#'\n",
    NULL
};

static const char *const cvswrappers_contents[] = {
    "# This file affects handling of files based on their names.\n",
    "#\n",
    "# The -t/-f options allow one to treat directories of files\n",
    "# as a single file, or to transform a file in other ways on\n",
    "# its way in and out of CVS.\n",
    "#\n",
    "# The -m option specifies whether CVS attempts to merge files.\n",
    "#\n",
    "# The -k option specifies keyword expansion (e.g. -kb for binary).\n",
    "#\n",
    "# Format of wrapper file ($CVSROOT/CVSROOT/cvswrappers or .cvswrappers)\n",
    "#\n",
    "#  wildcard     [option value][option value]...\n",
    "#\n",
    "#  where option is one of\n",
    "#  -f           from cvs filter     value: path to filter\n",
    "#  -t           to cvs filter       value: path to filter\n",
    "#  -m           update methodology  value: MERGE or COPY\n",
    "#  -k           expansion mode      value: b, o, kkv, &c\n",
    "#\n",
    "#  and value is a single-quote delimited value.\n",
    "# For example:\n",
    "#*.gif -k 'b'\n",
    NULL
};

static const char *const notify_contents[] = {
    "# The \"notify\" file controls where notifications from watches set by\n",
    "# \"cvs watch add\" or \"cvs edit\" are sent.  The first entry on a line is\n",
    "# a regular expression which is tested against the directory that the\n",
    "# change is being made to, relative to the $CVSROOT.  If it matches,\n",
    "# then the remainder of the line is a filter program that should contain\n",
    "# one occurrence of %s for the user to notify, and information on its\n",
    "# standard input.\n",
    "#\n",
    "# \"ALL\" or \"DEFAULT\" can be used in place of the regular expression.\n",
    "#\n",
    "# For example:\n",
    "#ALL mail %s -s \"CVS notification\"\n",
    NULL
};

static const char *const modules_contents[] = {
    "# Three different line formats are valid:\n",
    "# key    -a    aliases...\n",
    "# key [options] directory\n",
    "# key [options] directory files...\n",
    "#\n",
    "# Where \"options\" are composed of:\n",
    "# -i prog       Run \"prog\" on \"cvs commit\" from top-level of module.\n",
    "# -o prog       Run \"prog\" on \"cvs checkout\" of module.\n",
    "# -e prog       Run \"prog\" on \"cvs export\" of module.\n",
    "# -t prog       Run \"prog\" on \"cvs rtag\" of module.\n",
    "# -u prog       Run \"prog\" on \"cvs update\" of module.\n",
    "# -d dir        Place module in directory \"dir\" instead of module name.\n",
    "# -l            Top-level directory only -- do not recurse.\n",
    "#\n",
    "# NOTE:  If you change any of the \"Run\" options above, you'll have to\n",
    "# release and re-checkout any working directories of these modules.\n",
    "#\n",
    "# And \"directory\" is a path to a directory relative to $CVSROOT.\n",
    "#\n",
```

```
    "# The \"-a\" option specifies an alias.  An alias is interpreted as if\n",
    "# everything on the right of the \"-a\" had been typed on the command line.\n",
    "#\n",
    "# You can encode a module within a module by using the special '&'\n",
    "# character to interpose another module into the current module.  This\n",
    "# can be useful for creating a module that consists of many directories\n",
    "# spread out over the entire source repository.\n",
    NULL
};

static const char *const config_contents[] = {
    "# Set this to \"no\" if pserver shouldn't check system users/passwords\n",
    "#SystemAuth=no\n",
    "\n",
    "# Set `PreservePermissions' to `yes' to save file status information\n",
    "# in the repository.\n",
    "#PreservePermissions=no\n",
    "\n",
    "# Set `TopLevelAdmin' to `yes' to create a CVS directory at the top\n",
    "# level of the new working directory when using the `cvs checkout'\n",
    "# command.\n",
    "#TopLevelAdmin=no\n",
    NULL
};

static const struct admin_file filelist[] = {
    {CVSROOTADM_LOGINFO,
        "no logging of 'cvs commit' messages is done without a %s file",
        &loginfo_contents[0]},
    {CVSROOTADM_RCSINFO,
        "a %s file can be used to configure 'cvs commit' templates",
        rcsinfo_contents},
    {CVSROOTADM_EDITINFO,
        "a %s file can be used to validate log messages",
        editinfo_contents},
    {CVSROOTADM_VERIFYMSG,
        "a %s file can be used to validate log messages",
        verifymsg_contents},
    {CVSROOTADM_COMMITINFO,
        "a %s file can be used to configure 'cvs commit' checking",
        commitinfo_contents},
    {CVSROOTADM_TAGINFO,
        "a %s file can be used to configure 'cvs tag' checking",
        taginfo_contents},
    {CVSROOTADM_IGNORE,
        "a %s file can be used to specify files to ignore",
        NULL},
    {CVSROOTADM_CHECKOUTLIST,
        "a %s file can specify extra CVSROOT files to auto-checkout",
        checkoutlist_contents},
    {CVSROOTADM_WRAPPER,
        "a %s file can be used to specify files to treat as wrappers",
        cvswrappers_contents},
    {CVSROOTADM_NOTIFY,
        "a %s file can be used to specify where notifications go",
        notify_contents},
    {CVSROOTADM_MODULES,
        /* modules is special-cased in mkmodules.  */
        NULL,
        modules_contents},
    {CVSROOTADM_READERS,
        "a %s file specifies read-only users",
        NULL},
    {CVSROOTADM_WRITERS,
        "a %s file specifies read/write users",
        NULL},

    /* Some have suggested listing CVSROOTADM_PASSWD here too.  This
       would mean that CVS commands which operate on the
       CVSROOTADM_PASSWD file would transmit hashed passwords over the
       net.  This might seem to be no big deal, as pserver normally
       transmits cleartext passwords, but the difference is that
       CVSROOTADM_PASSWD contains *all* passwords, not just the ones
       currently being used.  For example, it could be too easy to
       accidentally give someone readonly access to CVSROOTADM_PASSWD
       (e.g. via anonymous CVS or cvsweb), and then if there are any
       guessable passwords for read/write access (usually there will be)
       they get read/write access.

       Another worry is the implications of storing old passwords--if
       someone used a password in the past they might be using it
       elsewhere, using a similar password, etc, and so saving old
       passwords, even hashed, is probably not a good idea.  */

    {CVSROOTADM_CONFIG,
        "a %s file configures various behaviors",
        config_contents},
    {NULL, NULL}
};
```

```
    /* Rebuild the checked out administrative files in directory DIR.  */
360 int
    mkmodules (dir)
        char *dir;
    {
        struct saved_cwd cwd;
        char *temp;
        char *cp, *last, *fname;
    #ifdef MY_NDBM
        DBM *db;
    #endif
370     FILE *fp;
        char *line = NULL;
        size_t line_allocated = 0;
        const struct admin_file *fileptr;

        if (save_cwd (&cwd))
            error_exit ();

        if ( CVS_CHDIR (dir) < 0)
            error (1, errno, "cannot chdir to %s", dir);
380
        /*
         * First, do the work necessary to update the "modules" database.
         */
        temp = make_tempfile ();
        switch (checkout_file (CVSROOTADM_MODULES, temp))
        {

            case 0:             /* everything ok */
    #ifdef MY_NDBM
390             /* open it, to generate any duplicate errors */
                if ((db = dbm_open (temp, O_RDONLY, 0666)) != NULL)
                    dbm_close (db);
    #else
                write_dbmfile (temp);
                rename_dbmfile (temp);
    #endif
                rename_rcsfile (temp, CVSROOTADM_MODULES);
                break;

400         case -1:                /* fork failed */
                (void) unlink_file (temp);
                error (1, errno, "cannot check out %s", CVSROOTADM_MODULES);
                /* NOTREACHED */

            default:
                error (0, 0,
                    "'cvs checkout' is less functional without a %s file",
                    CVSROOTADM_MODULES);
                break;
410     }                           /* switch on checkout_file() */

        (void) unlink_file (temp);
        free (temp);

        /* Checkout the files that need it in CVSROOT dir */
        for (fileptr = filelist; fileptr && fileptr->filename; fileptr++) {
            if (fileptr->errormsg == NULL)
                continue;
            temp = make_tempfile ();
420         if (checkout_file (fileptr->filename, temp) == 0)
                rename_rcsfile (temp, fileptr->filename);
    #if 0
            /*
             * If there was some problem other than the file not existing,
             * checkout_file already printed a real error message.  If the
             * file does not exist, it is harmless--it probably just means
             * that the repository was created with an old version of CVS
             * which didn't have so many files in CVSROOT.
             */
430         else if (fileptr->errormsg)
                error (0, 0, fileptr->errormsg, fileptr->filename);
    #endif
            (void) unlink_file (temp);
            free (temp);
        }

        fp = CVS_FOPEN (CVSROOTADM_CHECKOUTLIST, "r");
        if (fp)
        {
440         /*
             * File format:
             *  [<whitespace>]<filename><whitespace><error message><end-of-line>
             *
             * comment lines begin with '#'
             */
            while (getline (&line, &line_allocated, fp) >= 0)
            {
                /* skip lines starting with # */
```

```
            if (line[0] == '#')
                continue;

            if ((last = strrchr (line, '\n')) != NULL)
                *last = '\0';                   /* strip the newline */

            /* Skip leading white space. */
            for (fname = line; *fname && isspace(*fname); fname++)
                ;

            /* Find end of filename. */
            for (cp = fname; *cp && !isspace(*cp); cp++)
                ;
            *cp = '\0';

            temp = make_tempfile ();
            if (checkout_file (fname, temp) == 0)
            {
                rename_rcsfile (temp, fname);
            }
            else
            {
                for (cp++; cp < last && *last && isspace(*last); cp++)
                    ;
                if (cp < last && *cp)
                    error (0, 0, cp, fname);
            }
            free (temp);
        }
        if (line)
            free (line);
        if (ferror (fp))
            error (0, errno, "cannot read %s", CVSROOTADM_CHECKOUTLIST);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", CVSROOTADM_CHECKOUTLIST);
    }
    else
    {
        /* Error from CVS_FOPEN.  */
        if (!existence_error (errno))
            error (0, errno, "cannot open %s", CVSROOTADM_CHECKOUTLIST);
    }

    if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);

    return (0);
}

/*
 * Yeah, I know, there are NFS race conditions here.
 */
static char *
make_tempfile ()
{
    static int seed = 0;
    int fd;
    char *temp;

    if (seed == 0)
        seed = getpid ();
    temp = xmalloc (sizeof (BAKPREFIX) + 40);
    while (1)
    {
        (void) sprintf (temp, "%s%d", BAKPREFIX, seed++);
        if ((fd = CVS_OPEN (temp, O_CREAT|O_EXCL|O_RDWR, 0666)) != -1)
            break;
        if (errno != EEXIST)
            error (1, errno, "cannot create temporary file %s", temp);
    }
    if (close(fd) < 0)
        error(1, errno, "cannot close temporary file %s", temp);
    return temp;
}

static int
checkout_file (file, temp)
    char *file;
    char *temp;
{
    char *rcs;
    RCSNode *rcsnode;
    int retcode = 0;

    if (noexec)
        return 0;

    rcs = xmalloc (strlen (file) + 5);
    strcpy (rcs, file);
```

```
    strcat (rcs, RCSEXT);
    if (!isfile (rcs))
    {
        free (rcs);
        return (1);
    }
    rcsnode = RCS_parsercsfile (rcs);
    retcode = RCS_checkout (rcsnode, NULL, NULL, NULL, NULL, temp,
                            (RCSCHECKOUTPROC) NULL, (void *) NULL);
    if (retcode != 0)
    {
        error (0, 0, "failed to check out %s file",
               file);
    }
    freercsnode (&rcsnode);
    free (rcs);
    return (retcode);
}

#ifndef MY_NDBM

static void
write_dbmfile (temp)
    char *temp;
{
    char line[DBLKSIZ], value[DBLKSIZ];
    FILE *fp;
    DBM *db;
    char *cp, *vp;
    datum key, val;
    int len, cont, err = 0;

    fp = open_file (temp, "r");
    if ((db = dbm_open (temp, O_RDWR | O_CREAT | O_TRUNC, 0666)) == NULL)
        error (1, errno, "cannot open dbm file %s for creation", temp);
    for (cont = 0; fgets (line, sizeof (line), fp) != NULL;)
    {
        if ((cp = strrchr (line, '\n')) != NULL)
            *cp = '\0';             /* strip the newline */

        /*
         * Add the line to the value, at the end if this is a continuation
         * line; otherwise at the beginning, but only after any trailing
         * backslash is removed.
         */
        vp = value;
        if (cont)
            vp += strlen (value);

        /*
         * See if the line we read is a continuation line, and strip the
         * backslash if so.
         */
        len = strlen (line);
        if (len > 0)
            cp = &line[len - 1];
        else
            cp = line;
        if (*cp == '\\')
        {
            cont = 1;
            *cp = '\0';
        }
        else
        {
            cont = 0;
        }
        (void) strcpy (vp, line);
        if (value[0] == '#')
            continue;               /* comment line */
        vp = value;
        while (*vp && isspace (*vp))
            vp++;
        if (*vp == '\0')
            continue;               /* empty line */

        /*
         * If this was not a continuation line, add the entry to the database
         */
        if (!cont)
        {
            key.dptr = vp;
            while (*vp && !isspace (*vp))
                vp++;
            key.dsize = vp - key.dptr;
            *vp++ = '\0';           /* NULL terminate the key */
            while (*vp && isspace (*vp))
                vp++;               /* skip whitespace to value */
            if (*vp == '\0')
            {
```

```
                error (0, 0, "warning: NULL value for key `%s'", key.dptr);
                continue;
            }
            val.dptr = vp;
            val.dsize = strlen (vp);
            if (dbm_store (db, key, val, DBM_INSERT) == 1)
            {
                error (0, 0, "duplicate key found for `%s'", key.dptr);
                err++;
            }
        }
    }
    dbm_close (db);
    (void) fclose (fp);
    if (err)
    {
        char dotdir[50], dotpag[50], dotdb[50];

        (void) sprintf (dotdir, "%s.dir", temp);
        (void) sprintf (dotpag, "%s.pag", temp);
        (void) sprintf (dotdb, "%s.db", temp);
        (void) unlink_file (dotdir);
        (void) unlink_file (dotpag);
        (void) unlink_file (dotdb);
        error (1, 0, "DBM creation failed; correct above errors");
    }
}

static void
rename_dbmfile (temp)
    char *temp;
{
    char newdir[50], newpag[50], newdb[50];
    char dotdir[50], dotpag[50], dotdb[50];
    char bakdir[50], bakpag[50], bakdb[50];

    (void) sprintf (dotdir, "%s.dir", CVSROOTADM_MODULES);
    (void) sprintf (dotpag, "%s.pag", CVSROOTADM_MODULES);
    (void) sprintf (dotdb, "%s.db", CVSROOTADM_MODULES);
    (void) sprintf (bakdir, "%s%s.dir", BAKPREFIX, CVSROOTADM_MODULES);
    (void) sprintf (bakpag, "%s%s.pag", BAKPREFIX, CVSROOTADM_MODULES);
    (void) sprintf (bakdb, "%s%s.db", BAKPREFIX, CVSROOTADM_MODULES);
    (void) sprintf (newdir, "%s.dir", temp);
    (void) sprintf (newpag, "%s.pag", temp);
    (void) sprintf (newdb, "%s.db", temp);

    (void) chmod (newdir, 0666);
    (void) chmod (newpag, 0666);
    (void) chmod (newdb, 0666);

    /* don't mess with me */
    SIG_beginCrSect ();

    (void) unlink_file (bakdir);  /* rm .#modules.dir .#modules.pag */
    (void) unlink_file (bakpag);
    (void) unlink_file (bakdb);
    (void) CVS_RENAME (dotdir, bakdir); /* mv modules.dir .#modules.dir */
    (void) CVS_RENAME (dotpag, bakpag); /* mv modules.pag .#modules.pag */
    (void) CVS_RENAME (dotdb, bakdb); /* mv modules.db .#modules.db */
    (void) CVS_RENAME (newdir, dotdir); /* mv "temp".dir modules.dir */
    (void) CVS_RENAME (newpag, dotpag); /* mv "temp".pag modules.pag */
    (void) CVS_RENAME (newdb, dotdb); /* mv "temp".db modules.db */

    /* OK - make my day */
    SIG_endCrSect ();
}

#endif                          /* !MY_NDBM */

static void
rename_rcsfile (temp, real)
    char *temp;
    char *real;
{
    char *bak;
    struct stat statbuf;
    char *rcs;

    /* Set "x" bits if set in original. */
    rcs = xmalloc (strlen (real) + sizeof (RCSEXT) + 10);
    (void) sprintf (rcs, "%s%s", real, RCSEXT);
    statbuf.st_mode = 0; /* in case rcs file doesn't exist, but it should... */
    (void) CVS_STAT (rcs, &statbuf);
    free (rcs);

    if (chmod (temp, 0444 | (statbuf.st_mode & 0111)) < 0)
        error (0, errno, "warning: cannot chmod %s", temp);
    bak = xmalloc (strlen (real) + sizeof (BAKPREFIX) + 10);
    (void) sprintf (bak, "%s%s", BAKPREFIX, real);
    (void) unlink_file (bak);       /* rm .#loginfo */
```

```
    (void) CVS_RENAME (real, bak); /* mv loginfo .#loginfo */
    (void) CVS_RENAME (temp, real); /* mv "temp" loginfo */
    free (bak);
}

const char *const init_usage[] = {
    "Usage: %s %s\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
init (argc, argv)
    int argc;
    char **argv;
{
    /* Name of CVSROOT directory.  */
    char *adm;
    /* Name of this administrative file.  */
    char *info;
    /* Name of ,v file for this administrative file.  */
    char *info_v;
    /* Exit status.  */
    int err;

    const struct admin_file *fileptr;

    umask (cvsumask);

    if (argc == -1 || argc > 1)
        usage (init_usage);

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();

        ign_setup ();
        send_init_command ();
        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    /* Note: we do *not* create parent directories as needed like the
       old cvsinit.sh script did.  Few utilities do that, and a
       non-existent parent directory is as likely to be a typo as something
       which needs to be created.  */
    mkdir_if_needed (CVSroot_directory);

    adm = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM) + 10);
    strcpy (adm, CVSroot_directory);
    strcat (adm, "/");
    strcat (adm, CVSROOTADM);
    mkdir_if_needed (adm);

    /* This is needed because we pass "fileptr->filename" not "info"
       to add_rcs_file below.  I think this would be easy to change,
       thus nuking the need for CVS_CHDIR here, but I haven't looked
       closely (e.g. see wrappers calls within add_rcs_file).  */
    if ( CVS_CHDIR (adm) < 0)
        error (1, errno, "cannot change to directory %s", adm);

    /* 80 is long enough for all the administrative file names, plus
       "/" and so on.  */
    info = xmalloc (strlen (adm) + 80);
    info_v = xmalloc (strlen (adm) + 80);
    for (fileptr = filelist; fileptr && fileptr->filename; ++fileptr)
    {
        if (fileptr->contents == NULL)
            continue;
        strcpy (info, adm);
        strcat (info, "/");
        strcat (info, fileptr->filename);
        strcpy (info_v, info);
        strcat (info_v, RCSEXT);
        if (isfile (info_v))
            /* We will check out this file in the mkmodules step.
               Nothing else is required.  */
            ;
        else
        {
            int retcode;

            if (!isfile (info))
            {
                FILE *fp;
                const char * const *p;

                fp = open_file (info, "w");
                for (p = fileptr->contents; *p != NULL; ++p)
```

```
                if (fputs (*p, fp) < 0)
                    error (1, errno, "cannot write %s", info);
            if (fclose (fp) < 0)
                error (1, errno, "cannot close %s", info);
        }
        /* The message used to say " of " and fileptr->filename after
           "initial checkin" but I fail to see the point as we know what
           file it is from the name.  */
        retcode = add_rcs_file ("initial checkin", info_v,
                                fileptr->filename, "1.1", NULL,
                                /* No vendor branch.  */
                                NULL, NULL, 0, NULL,
                                NULL, 0, NULL, NULL);
        if (retcode != 0)
            /* add_rcs_file already printed an error message.  */
            err = 1;
    }
}

/* Turn on history logging by default.  The user can remove the file
   to disable it.  */
strcpy (info, adm);
strcat (info, "/");
strcat (info, CVSROOTADM_HISTORY);
if (!isfile (info))
{
    FILE *fp;

    fp = open_file (info, "w");
    if (fclose (fp) < 0)
        error (1, errno, "cannot close %s", info);
}

free (info);
free (info_v);

mkmodules (adm);

free (adm);
return 0;
}
```

A.38 modules.c

```
/*
 *    Copyright (c) 1992, Brian Berliner and Jeff Polk
 *    Copyright (c) 1989-1992, Brian Berliner
 *
 *    You may distribute under the terms of the GNU General Public License
 *    as specified in the README file that comes with the CVS source distribution.
 *
 * Modules
 *
 *      Functions for accessing the modules file.
 *
 *      The modules file supports basically three formats of lines:
 *              key [options] directory files... [ -x directory [files] ] ...
 *              key [options] directory [ -x directory [files] ] ...
 *              key -a aliases...
 *
 *      The -a option allows an aliasing step in the parsing of the modules
 *      file.  The "aliases" listed on a line following the -a are
 *      processed one-by-one, as if they were specified as arguments on the
 *      command line.
 */

#include "cvs.h"
#include "savecwd.h"


/* Defines related to the syntax of the modules file.  */

/* Options in modules file.  Note that it is OK to use GNU getopt features;
   we already are arranging to make sure we are using the getopt distributed
   with CVS.  */
#define CVSMODULE_OPTS "+ad:i:lo:e:s:t:u:"

/* Special delimiter.  */
#define CVSMODULE_SPEC '&'

struct sortrec
{
    /* Name of the module, malloc'd.  */
    char *modname;
    /* If Status variable is set, this is either def_status or the malloc'd
       name of the status.  If Status is not set, the field is left
       uninitialized.  */
    char *status;
    /* Pointer to a malloc'd array which contains (1) the raw contents
       of the options and arguments, excluding comments, (2) a '\0',
       and (3) the storage for the "comment" field.  */
    char *rest;
    char *comment;
};

static int sort_order PROTO((const PTR l, const PTR r));
static void save_d PROTO((char *k, int ks, char *d, int ds));


/*
 * Open the modules file, and die if the CVSROOT environment variable
 * was not set.  If the modules file does not exist, that's fine, and
 * a warning message is displayed and a NULL is returned.
 */
DBM *
open_module ()
{
    char *mfile;
    DBM *retval;

    if (CVSroot_original == NULL)
    {
        error (0, 0, "must set the CVSROOT environment variable");
        error (1, 0, "or specify the '-d' global option");
    }
    mfile = xmalloc (strlen (CVSroot_directory) + sizeof (CVSROOTADM)
                     + sizeof (CVSROOTADM_MODULES) + 20);
    (void) sprintf (mfile, "%s/%s/%s", CVSroot_directory,
                    CVSROOTADM, CVSROOTADM_MODULES);
    retval = dbm_open (mfile, O_RDONLY, 0666);
    free (mfile);
    return retval;
}

/*
 * Close the modules file, if the open succeeded, that is
 */
void
close_module (db)
    DBM *db;
{
    if (db != NULL)
```

```
        dbm_close (db);
}

/*
 * This is the recursive function that processes a module name.
 * It calls back the passed routine for each directory of a module
 * It runs the post checkout or post tag proc from the modules file
 */
int
do_module (db, mname, m_type, msg, callback_proc, where,
           shorten, local_specified, run_module_prog, extra_arg)
    DBM *db;
    char *mname;
    enum mtype m_type;
    char *msg;
    CALLBACKPROC callback_proc;
    char *where;
    int shorten;
    int local_specified;
    int run_module_prog;
    char *extra_arg;
{
    char *checkin_prog = NULL;
    char *checkout_prog = NULL;
    char *export_prog = NULL;
    char *tag_prog = NULL;
    char *update_prog = NULL;
    struct saved_cwd cwd;
    int cwd_saved = 0;
    char *line;
    int modargc;
    int xmodargc;
    char **modargv;
    char **xmodargv;
    char *value;
    char *zvalue = NULL;
    char *mwhere = NULL;
    char *mfile = NULL;
    char *spec_opt = NULL;
    char *xvalue = NULL;
    int alias = 0;
    datum key, val;
    char *cp;
    int c, err = 0;
    int nonalias_opt = 0;

#ifdef SERVER_SUPPORT
    int restore_server_dir = 0;
    char *server_dir_to_restore = NULL;
    if (trace)
    {
        char *buf;

        /* We use cvs_outerr, rather than fprintf to stderr, because
           this may be called by server code with error_use_protocol
           set.  */
        buf = xmalloc (100
                       + strlen (mname)
                       + strlen (msg)
                       + (where ? strlen (where) : 0)
                       + (extra_arg ? strlen (extra_arg) : 0));
        sprintf (buf, "%c-> do_module (%s, %s, %s, %s)\n",
                 (server_active) ? 'S' : ' ',
                 mname, msg, where ? where : "",
                 extra_arg ? extra_arg : "");
        cvs_outerr (buf, 0);
        free (buf);
    }
#endif

    /* if this is a directory to ignore, add it to that list */
    if (mname[0] == '!' && mname[1] != '\0')
    {
        ign_dir_add (mname+1);
        goto do_module_return;
    }

    /* strip extra stuff from the module name */
    strip_trailing_slashes (mname);

    /*
     * Look up the module using the following scheme:
     *  1) look for mname as a module name
     *  2) look for mname as a directory
     *  3) look for mname as a file
     *  4) take mname up to the first slash and look it up as a module name
     *     (this is for checking out only part of a module)
     */

    /* look it up as a module name */
```

```
    key.dptr = mname;
180 key.dsize = strlen (key.dptr);
    if (db != NULL)
        val = dbm_fetch (db, key);
    else
        val.dptr = NULL;
    if (val.dptr != NULL)
    {
        /* null terminate the value  XXX - is this space ours? */
        val.dptr[val.dsize] = '\0';

190     /* If the line ends in a comment, strip it off */
        if ((cp = strchr (val.dptr, '#')) != NULL)
        {
            do
                *cp-- = '\0';
            while (isspace (*cp));
        }
        else
        {
            /* Always strip trailing spaces */
200         cp = strchr (val.dptr, '\0');
            while (cp > val.dptr && isspace(*--cp))
                *cp = '\0';
        }

        value = val.dptr;
        mwhere = xstrdup (mname);
        goto found;
    }
    else
210 {
        char *file;
        char *attic_file;
        char *acp;
        int is_found = 0;

        /* check to see if mname is a directory or file */
        file = xmalloc (strlen (CVSroot_directory) + strlen (mname) + 10);
        (void) sprintf (file, "%s/%s", CVSroot_directory, mname);
        attic_file = xmalloc (strlen (CVSroot_directory) + strlen (mname)
220                       + sizeof (CVSATTIC) + sizeof (RCSEXT) + 15);
        if ((acp = strrchr (mname, '/')) != NULL)
        {
            *acp = '\0';
            (void) sprintf (attic_file, "%s/%s/%s/%s%s", CVSroot_directory,
                            mname, CVSATTIC, acp + 1, RCSEXT);
            *acp = '/';
        }
        else
            (void) sprintf (attic_file, "%s/%s/%s%s", CVSroot_directory,
230                         CVSATTIC, mname, RCSEXT);

        if (isdir (file))
        {
            value = mname;
            is_found = 1;
        }
        else
        {
            (void) strcat (file, RCSEXT);
240         if (isfile (file) || isfile (attic_file))
            {
                /* if mname was a file, we have to split it into "dir file" */
                if ((cp = strrchr (mname, '/')) != NULL && cp != mname)
                {
                    char *slashp;

                    /* put the ' ' in a copy so we don't mess up the
                       original */
                    xvalue = xmalloc (strlen (mname) + 2);
250                 value = strcpy (xvalue, mname);
                    slashp = strrchr (value, '/');
                    *slashp = ' ';
                }
                else
                {
                    /*
                     * the only '/' at the beginning or no '/' at all
                     * means the file we are interested in is in CVSROOT
                     * itself so the directory should be '.'
260                  */
                    if (cp == mname)
                    {
                        /* drop the leading / if specified */
                        xvalue = xmalloc (strlen (mname) + 10);
                        value = strcpy (xvalue, ". ");
                        (void) strcat (xvalue, mname + 1);
                    }
                    else
```

```
                    {
                        /* otherwise just copy it */
                        xvalue = xmalloc (strlen (mname) + 10);
                        value = strcpy (xvalue, ". ");
                        (void) strcat (xvalue, mname);
                    }
                }
                is_found = 1;
            }
            else
            {
                /* This initialization suppresses a warning from gcc -Wall.  */
                value = NULL;
            }
        }
        free (attic_file);
        free (file);

        if (is_found)
            goto found;
    }

    /* look up everything to the first / as a module */
    if (mname[0] != '/' && (cp = strchr (mname, '/')) != NULL)
    {
        /* Make the slash the new end of the string temporarily */
        *cp = '\0';
        key.dptr = mname;
        key.dsize = strlen (key.dptr);

        /* do the lookup */
        if (db != NULL)
            val = dbm_fetch (db, key);
        else
            val.dptr = NULL;

        /* if we found it, clean up the value and life is good */
        if (val.dptr != NULL)
        {
            char *cp2;

            /* null terminate the value XXX - is this space ours? */
            val.dptr[val.dsize] = '\0';

            /* If the line ends in a comment, strip it off */
            if ((cp2 = strchr (val.dptr, '#')) != NULL)
            {
                do
                    *cp2-- = '\0';
                while (isspace (*cp2));
            }
            value = val.dptr;

            /* mwhere gets just the module name */
            mwhere = xstrdup (mname);
            mfile = cp + 1;

            /* put the / back in mname */
            *cp = '/';

            goto found;
        }

        /* put the / back in mname */
        *cp = '/';
    }

    /* if we got here, we couldn't find it using our search, so give up */
    error (0, 0, "cannot find module `%s' - ignored", mname);
    err++;
    goto do_module_return;


    /*
     * At this point, we found what we were looking for in one
     * of the many different forms.
     */
found:

    /* remember where we start */
    if (save_cwd (&cwd))
        error_exit ();
    cwd_saved = 1;

    /* copy value to our own string since if we go recursive we'll be
       really screwed if we do another dbm lookup */
    zvalue = xstrdup (value);
    value = zvalue;

    /* search the value for the special delimiter and save for later */
```

```
    if ((cp = strchr (value, CVSMODULE_SPEC)) != NULL)
360 {
        *cp = '\0';                 /* null out the special char */
        spec_opt = cp + 1;          /* save the options for later */

        if (cp != value)            /* strip whitespace if necessary */
            while (isspace (*--cp))
                *cp = '\0';

        if (cp == value)
        {
370         /*
             * we had nothing but special options, so skip arg
             * parsing and regular stuff entirely
             *
             * If there were only special ones though, we must
             * make the appropriate directory and cd to it
             */
            char *dir;

            /* XXX - XXX - MAJOR HACK - DO NOT SHIP - this needs to
380            be !pipeout, but we don't know that here yet */
            if (!run_module_prog)
                goto out;

            dir = where ? where : mname;
            /* XXX - think about making null repositories at each dir here
                    instead of just at the bottom */
            make_directories (dir);
            if ( CVS_CHDIR (dir) < 0)
            {
390             error (0, errno, "cannot chdir to %s", dir);
                spec_opt = NULL;
                err++;
                goto out;
            }
            if (!isfile (CVSADM))
            {
                char *nullrepos;

                nullrepos = emptydir_name ();
400
                Create_Admin (".", dir,
                              nullrepos, (char *) NULL, (char *) NULL, 0, 0);
                if (!noexec)
                {
                    FILE *fp;

                    fp = open_file (CVSADM_ENTSTAT, "w+");
                    if (fclose (fp) == EOF)
                        error (1, errno, "cannot close %s", CVSADM_ENTSTAT);
410 #ifdef SERVER_SUPPORT
                    if (server_active)
                        server_set_entstat (dir, nullrepos);
    #endif
                }
                free (nullrepos);
            }
          out:
            goto do_special;
        }
420 }

    /* don't do special options only part of a module was specified */
    if (mfile != NULL)
        spec_opt = NULL;

    /*
     * value now contains one of the following:
     *    1) dir
     *    2) dir file
430  *    3) the value from modules without any special args
     *            [ args ] dir [file] [file] ...
     *       or     -a module [ module ] ...
     */

    /* Put the value on a line with XXX prepended for getopt to eat */
    line = xmalloc (strlen (value) + 10);
    (void) sprintf (line, "%s %s", "XXX", value);

    /* turn the line into an argv[] array */
440 line2argv (&xmodargc, &xmodargv, line, " \t");
    free (line);
    modargc = xmodargc;
    modargv = xmodargv;

    /* parse the args */
    optind = 0;
    while ((c = getopt (modargc, modargv, CVSMODULE_OPTS)) != -1)
    {
```

```
        switch (c)
        {
            case 'a':
                alias = 1;
                break;
            case 'd':
                nonalias_opt = 1;
                if (mwhere)
                    free (mwhere);
                mwhere = xstrdup (optarg);
                break;
            case 'i':
                nonalias_opt = 1;
                if (checkin_prog)
                    free (checkin_prog);
                checkin_prog = xstrdup (optarg);
                break;
            case 'l':
                nonalias_opt = 1;
                local_specified = 1;
                break;
            case 'o':
                nonalias_opt = 1;
                if (checkout_prog)
                    free (checkout_prog);
                checkout_prog = xstrdup (optarg);
                break;
            case 'e':
                nonalias_opt = 1;
                if (export_prog)
                    free (export_prog);
                export_prog = xstrdup (optarg);
                break;
            case 't':
                nonalias_opt = 1;
                if (tag_prog)
                    free (tag_prog);
                tag_prog = xstrdup (optarg);
                break;
            case 'u':
                nonalias_opt = 1;
                if (update_prog)
                    free (update_prog);
                update_prog = xstrdup (optarg);
                break;
            case '?':
                error (0, 0,
                       "modules file has invalid option for key %s value %s",
                       key.dptr, val.dptr);
                err++;
                goto do_module_return;
        }
    }
    modargc -= optind;
    modargv += optind;
    if (modargc == 0)
    {
        error (0, 0, "modules file missing directory for module %s", mname);
        ++err;
        goto do_module_return;
    }

    if (alias && nonalias_opt)
    {
        /* The documentation has never said it is legal to specify
           -a along with another option.  And I believe that in the past
           CVS has ignored the options other than -a, more or less, in this
           situation.  */
        error (0, 0, "\
-a cannot be specified in the modules file along with other options");
        ++err;
        goto do_module_return;
    }

    /* if this was an alias, call ourselves recursively for each module */
    if (alias)
    {
        int i;

        for (i = 0; i < modargc; i++)
        {
            if (strcmp (mname, modargv[i]) == 0)
                error (0, 0,
                       "module `%s' in modules file contains infinite loop",
                       mname);
            else
                err += do_module (db, modargv[i], m_type, msg, callback_proc,
                                  where, shorten, local_specified,
                                  run_module_prog, extra_arg);
        }
```

```
        goto do_module_return;
    }

    if (mfile != NULL && modargc > 1)
    {
        error (0, 0, "\
module `%s' is a request for a file in a module which is not a directory",
               mname);
        ++err;
        goto do_module_return;
    }

    /* otherwise, process this module */
    err += callback_proc (&modargc, modargv, where, mwhere, mfile, shorten,
                          local_specified, mname, msg);

    free_names (&xmodargc, xmodargv);

    /* if there were special include args, process them now */

  do_special:

    /* blow off special options if -l was specified */
    if (local_specified)
        spec_opt = NULL;

#ifdef SERVER_SUPPORT
    /* We want to check out into the directory named by the module.
       So we set a global variable which tells the server to glom that
       directory name onto the front.  A cleaner approach would be some
       way of passing it down to the recursive call, through the
       callback_proc, to start_recursion, and then into the update_dir in
       the struct file_info.  That way the "Updating foo" message could
       print the actual directory we are checking out into.

       For local CVS, this is handled by the chdir call above
       (directly or via the callback_proc).  */
    if (server_active && spec_opt != NULL)
    {
        char *change_to;

        change_to = where ? where : (mwhere ? mwhere : mname);
        server_dir_to_restore = server_dir;
        restore_server_dir = 1;
        server_dir =
            xmalloc ((server_dir_to_restore != NULL
                      ? strlen (server_dir_to_restore)
                      : 0)
                     + strlen (change_to)
                     + 5);
        server_dir[0] = '\0';
        if (server_dir_to_restore != NULL)
        {
            strcat (server_dir, server_dir_to_restore);
            strcat (server_dir, "/");
        }
        strcat (server_dir, change_to);
    }
#endif

    while (spec_opt != NULL)
    {
        char *next_opt;

        cp = strchr (spec_opt, CVSMODULE_SPEC);
        if (cp != NULL)
        {
            /* save the beginning of the next arg */
            next_opt = cp + 1;

            /* strip whitespace off the end */
            do
                *cp = '\0';
            while (isspace (*--cp));
        }
        else
            next_opt = NULL;

        /* strip whitespace from front */
        while (isspace (*spec_opt))
            spec_opt++;

        if (*spec_opt == '\0')
            error (0, 0, "Mal-formed %c option for module %s - ignored",
                   CVSMODULE_SPEC, mname);
        else
            err += do_module (db, spec_opt, m_type, msg, callback_proc,
                              (char *) NULL, 0, local_specified,
                              run_module_prog, extra_arg);
        spec_opt = next_opt;
```

```
    }

#ifdef SERVER_SUPPORT
    if (server_active && restore_server_dir)
    {
        free (server_dir);
        server_dir = server_dir_to_restore;
    }
#endif

    /* write out the checkin/update prog files if necessary */
#ifdef SERVER_SUPPORT
    if (err == 0 && !noexec && m_type == CHECKOUT && server_expanding)
    {
        if (checkin_prog != NULL)
            server_prog (where ? where : mname, checkin_prog, PROG_CHECKIN);
        if (update_prog != NULL)
            server_prog (where ? where : mname, update_prog, PROG_UPDATE);
    }
    else
#endif
    if (err == 0 && !noexec && m_type == CHECKOUT && run_module_prog)
    {
        FILE *fp;

        if (checkin_prog != NULL)
        {
            fp = open_file (CVSADM_CIPROG, "w+");
            (void) fprintf (fp, "%s\n", checkin_prog);
            if (fclose (fp) == EOF)
                error (1, errno, "cannot close %s", CVSADM_CIPROG);
        }
        if (update_prog != NULL)
        {
            fp = open_file (CVSADM_UPROG, "w+");
            (void) fprintf (fp, "%s\n", update_prog);
            if (fclose (fp) == EOF)
                error (1, errno, "cannot close %s", CVSADM_UPROG);
        }
    }

    /* cd back to where we started */
    if (restore_cwd (&cwd, NULL))
        error_exit ();
    free_cwd (&cwd);
    cwd_saved = 0;

    /* run checkout or tag prog if appropriate */
    if (err == 0 && run_module_prog)
    {
        if ((m_type == TAG && tag_prog != NULL) ||
            (m_type == CHECKOUT && checkout_prog != NULL) ||
            (m_type == EXPORT && export_prog != NULL))
        {
            /*
             * If a relative pathname is specified as the checkout, tag
             * or export proc, try to tack on the current "where" value.
             * if we can't find a matching program, just punt and use
             * whatever is specified in the modules file.
             */
            char *real_prog = NULL;
            char *prog = (m_type == TAG ? tag_prog :
                          (m_type == CHECKOUT ? checkout_prog : export_prog));
            char *real_where = (where != NULL ? where : mwhere);
            char *expanded_path;

            if ((*prog != '/') && (*prog != '.'))
            {
                real_prog = xmalloc (strlen (real_where) + strlen (prog)
                                     + 10);
                (void) sprintf (real_prog, "%s/%s", real_where, prog);
                if (isfile (real_prog))
                    prog = real_prog;
            }

            /* XXX can we determine the line number for this entry??? */
            expanded_path = expand_path (prog, "modules", 0);
            if (expanded_path != NULL)
            {
                run_setup (expanded_path);
                run_arg (real_where);

                if (extra_arg)
                    run_arg (extra_arg);

                if (!quiet)
                {
                    cvs_output (program_name, 0);
                    cvs_output (" ", 1);
                    cvs_output (command_name, 0);
```

```
                cvs_output (": Executing '", 0);
                run_print (stdout);
                cvs_output ("'\n", 0);
            }
            err += run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
            free (expanded_path);
        }
        free (real_prog);
    }
}

do_module_return:
    /* clean up */
    if (mwhere)
        free (mwhere);
    if (checkin_prog)
        free (checkin_prog);
    if (checkout_prog)
        free (checkout_prog);
    if (export_prog)
        free (export_prog);
    if (tag_prog)
        free (tag_prog);
    if (update_prog)
        free (update_prog);
    if (cwd_saved)
        free_cwd (&cwd);
    if (zvalue != NULL)
        free (zvalue);

    if (xvalue != NULL)
        free (xvalue);
    return (err);
}

/* - Read all the records from the modules database into an array.
   - Sort the array depending on what format is desired.
   - Print the array in the format desired.

   Currently, there are only two "desires":

   1. Sort by module name and format the whole entry including switches,
      files and the comment field: (Including aliases)

      modulename   -s switches, one per line, even if
                   -i it has many switches.
                   Directories and files involved, formatted
                   to cover multiple lines if necessary.
                   # Comment, also formatted to cover multiple
                   # lines if necessary.

   2. Sort by status field string and print:  (*not* including aliases)

      modulename    STATUS Directories and files involved, formatted
                           to cover multiple lines if necessary.
                           # Comment, also formatted to cover multiple
                           # lines if necessary.
*/

static struct sortrec *s_head;

static int s_max = 0;           /* Number of elements allocated */
static int s_count = 0;         /* Number of elements used */

static int Status;              /* Nonzero if the user is
                                   interested in status
                                   information as well as
                                   module name */
static char def_status[] = "NONE";

/* Sort routine for qsort:
   - If we want the "Status" field to be sorted, check it first.
   - Then compare the "module name" fields.  Since they are unique, we don't
     have to look further.
*/
static int
sort_order (l, r)
    const PTR l;
    const PTR r;
{
    int i;
    const struct sortrec *left = (const struct sortrec *) l;
    const struct sortrec *right = (const struct sortrec *) r;

    if (Status)
    {
        /* If Sort by status field, compare them. */
        if ((i = strcmp (left->status, right->status)) != 0)
            return (i);
    }
```

```
    return (strcmp (left->modname, right->modname));
}

static void
save_d (k, ks, d, ds)
    char *k;
    int ks;
    char *d;
    int ds;
{
    char *cp, *cp2;
    struct sortrec *s_rec;

    if (Status && *d == '-' && *(d + 1) == 'a')
        return;                     /* We want "cvs co -s" and it is an alias! */

    if (s_count == s_max)
    {
        s_max += 64;
        s_head = (struct sortrec *) xrealloc ((char *) s_head, s_max * sizeof (*s_head));
    }
    s_rec = &s_head[s_count];
    s_rec->modname = cp = xmalloc (ks + 1);
    (void) strncpy (cp, k, ks);
    *(cp + ks) = '\0';

    s_rec->rest = cp2 = xmalloc (ds + 1);
    cp = d;
    *(cp + ds) = '\0'; /* Assumes an extra byte at end of static dbm buffer */

    while (isspace (*cp))
        cp++;
    /* Turn <spaces> into one ' ' - makes the rest of this routine simpler */
    while (*cp)
    {
        if (isspace (*cp))
        {
            *cp2++ = ' ';
            while (isspace (*cp))
                cp++;
        }
        else
            *cp2++ = *cp++;
    }
    *cp2 = '\0';

    /* Look for the "-s statusvalue" text */
    if (Status)
    {
        s_rec->status = def_status;

        for (cp = s_rec->rest; (cp2 = strchr (cp, '-')) != NULL; cp = ++cp2)
        {
            if (*(cp2 + 1) == 's' && *(cp2 + 2) == ' ')
            {
                char *status_start;

                cp2 += 3;
                status_start = cp2;
                while (*cp2 != ' ' && *cp2 != '\0')
                    cp2++;
                s_rec->status = xmalloc (cp2 - status_start + 1);
                strncpy (s_rec->status, status_start, cp2 - status_start);
                s_rec->status[cp2 - status_start] = '\0';
                cp = cp2;
                break;
            }
        }
    }
    else
        cp = s_rec->rest;

    /* Find comment field, clean up on all three sides & compress blanks */
    if ((cp2 = cp = strchr (cp, '#')) != NULL)
    {
        if (*--cp2 == ' ')
            *cp2 = '\0';
        if (*++cp == ' ')
            cp++;
        s_rec->comment = cp;
    }
    else
        s_rec->comment = "";

    s_count++;
}

/* Print out the module database as we know it.  If STATUS is
   non-zero, print out status information for each module. */
```

```
void
cat_module (status)
    int status;
{
    DBM *db;
    datum key, val;
    int i, c, wid, argc, cols = 80, indent, fill;
    int moduleargc;
    struct sortrec *s_h;
    char *cp, *cp2, **argv;
    char **moduleargv;

    Status = status;

    /* Read the whole modules file into allocated records */
    if (!(db = open_module ()))
        error (1, 0, "failed to open the modules file");

    for (key = dbm_firstkey (db); key.dptr != NULL; key = dbm_nextkey (db))
    {
        val = dbm_fetch (db, key);
        if (val.dptr != NULL)
            save_d (key.dptr, key.dsize, val.dptr, val.dsize);
    }

    close_module (db);

    /* Sort the list as requested */
    qsort ((PTR) s_head, s_count, sizeof (struct sortrec), sort_order);

    /*
     * Run through the sorted array and format the entries
     * indent = space for modulename + space for status field
     */
    indent = 12 + (status * 12);
    fill = cols - (indent + 2);
    for (s_h = s_head, i = 0; i < s_count; i++, s_h++)
    {
        char *line;

        /* Print module name (and status, if wanted) */
        line = xmalloc (strlen (s_h->modname) + 15);
        sprintf (line, "%-12s", s_h->modname);
        cvs_output (line, 0);
        free (line);
        if (status)
        {
            line = xmalloc (strlen (s_h->status) + 15);
            sprintf (line, " %-11s", s_h->status);
            cvs_output (line, 0);
            free (line);
        }

        line = xmalloc (strlen (s_h->modname) + strlen (s_h->rest) + 15);
        /* Parse module file entry as command line and print options */
        (void) sprintf (line, "%s %s", s_h->modname, s_h->rest);
        line2argv (&moduleargc, &moduleargv, line, " \t");
        free (line);
        argc = moduleargc;
        argv = moduleargv;

        optind = 0;
        wid = 0;
        while ((c = getopt (argc, argv, CVSMODULE_OPTS)) != -1)
        {
            if (!status)
            {
                if (c == 'a' || c == 'l')
                {
                    char buf[5];

                    sprintf (buf, " -%c", c);
                    cvs_output (buf, 0);
                    wid += 3;       /* Could just set it to 3 */
                }
                else
                {
                    char buf[10];

                    if (strlen (optarg) + 4 + wid > (unsigned) fill)
                    {
                        int j;

                        cvs_output ("\n", 1);
                        for (j = 0; j < indent; ++j)
                            cvs_output (" ", 1);
                        wid = 0;
                    }
                    sprintf (buf, " -%c ", c);
                    cvs_output (buf, 0);
```

```
                cvs_output (optarg, 0);
                wid += strlen (optarg) + 4;
            }
        }
    }
    argc -= optind;
    argv += optind;

    /* Format and Print all the files and directories */
    for (; argc--; argv++)
    {
        if (strlen (*argv) + wid > (unsigned) fill)
        {
            int j;

            cvs_output ("\n", 1);
            for (j = 0; j < indent; ++j)
                cvs_output (" ", 1);
            wid = 0;
        }
        cvs_output (" ", 1);
        cvs_output (*argv, 0);
        wid += strlen (*argv) + 1;
    }
    cvs_output ("\n", 1);

    /* Format the comment field - save_d (), compressed spaces */
    for (cp2 = cp = s_h->comment; *cp; cp2 = cp)
    {
        int j;

        for (j = 0; j < indent; ++j)
            cvs_output (" ", 1);
        cvs_output (" # ", 0);
        if (strlen (cp2) < (unsigned) (fill - 2))
        {
            cvs_output (cp2, 0);
            cvs_output ("\n", 1);
            break;
        }
        cp += fill - 2;
        while (*cp != ' ' && cp > cp2)
            cp--;
        if (cp == cp2)
        {
            cvs_output (cp2, 0);
            cvs_output ("\n", 1);
            break;
        }

        *cp++ = '\0';
        cvs_output (cp2, 0);
        cvs_output ("\n", 1);
    }

    free_names(&moduleargc, moduleargv);
    /* FIXME-leak: here is where we would free s_h->modname, s_h->rest,
       and if applicable, s_h->status.  Not exactly a memory leak,
       in the sense that we are about to exit(), but may be worth
       noting if we ever do a multithreaded server or something of
       the sort.  */
}
/* FIXME-leak: as above, here is where we would free s_head.  */
}
```

A.39 myndbm.c

```
/*
 * Copyright (c) 1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * A simple ndbm-emulator for CVS.  It parses a text file of the format:
 *
 * key value
 *
 * at dbm_open time, and loads the entire file into memory.  As such, it is
 * probably only good for fairly small modules files.  Ours is about 30K in
 * size, and this code works fine.
 */

#include <assert.h>
#include "cvs.h"
#include "getline.h"

#ifdef MY_NDBM

static void mydbm_load_file PROTO ((FILE *, List *));

/* Returns NULL on error in which case errno has been set to indicate
   the error.  Can also call error() itself.  */
/* ARGSUSED */
DBM *
mydbm_open (file, flags, mode)
    char *file;
    int flags;
    int mode;
{
    FILE *fp;
    DBM *db;

    fp = CVS_FOPEN (file, FOPEN_BINARY_READ);
    if (fp == NULL && !(existence_error (errno) && (flags & O_CREAT)))
        return ((DBM *) 0);

    db = (DBM *) xmalloc (sizeof (*db));
    db->dbm_list = getlist ();
    db->modified = 0;
    db->name = xstrdup (file);

    if (fp != NULL)
    {
        mydbm_load_file (fp, db->dbm_list);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", file);
    }
    return (db);
}

static int write_item PROTO ((Node *, void *));

static int
write_item (node, data)
    Node *node;
    void *data;
{
    FILE *fp = (FILE *)data;
    fputs (node->key, fp);
    fputs (" ", fp);
    fputs (node->data, fp);
    fputs ("\012", fp);
    return 0;
}

void
mydbm_close (db)
    DBM *db;
{
    if (db->modified)
    {
        FILE *fp;
        fp = CVS_FOPEN (db->name, FOPEN_BINARY_WRITE);
        if (fp == NULL)
            error (1, errno, "cannot write %s", db->name);
        walklist (db->dbm_list, write_item, (void *)fp);
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", db->name);
    }
    free (db->name);
    dellist (&db->dbm_list);
    free ((char *) db);
}

datum
```

```
mydbm_fetch (db, key)
    DBM *db;
    datum key;
{
    Node *p;
    char *s;
    datum val;

    /* make sure it's null-terminated */
    s = xmalloc (key.dsize + 1);
    (void) strncpy (s, key.dptr, key.dsize);
    s[key.dsize] = '\0';

    p = findnode (db->dbm_list, s);
    if (p)
    {
        val.dptr = p->data;
        val.dsize = strlen (p->data);
    }
    else
    {
        val.dptr = (char *) NULL;
        val.dsize = 0;
    }
    free (s);
    return (val);
}

datum
mydbm_firstkey (db)
    DBM *db;
{
    Node *head, *p;
    datum key;

    head = db->dbm_list->list;
    p = head->next;
    if (p != head)
    {
        key.dptr = p->key;
        key.dsize = strlen (p->key);
    }
    else
    {
        key.dptr = (char *) NULL;
        key.dsize = 0;
    }
    db->dbm_next = p->next;
    return (key);
}

datum
mydbm_nextkey (db)
    DBM *db;
{
    Node *head, *p;
    datum key;

    head = db->dbm_list->list;
    p = db->dbm_next;
    if (p != head)
    {
        key.dptr = p->key;
        key.dsize = strlen (p->key);
    }
    else
    {
        key.dptr = (char *) NULL;
        key.dsize = 0;
    }
    db->dbm_next = p->next;
    return (key);
}

/* Note: only updates the in-memory copy, which is written out at
   mydbm_close time.  Note: Also differs from DBM in that on duplication,
   it gives a warning, rather than either DBM_INSERT or DBM_REPLACE
   behavior.  */
int
mydbm_store (db, key, value, flags)
    DBM *db;
    datum key;
    datum value;
    int flags;
{
    Node *node;

    node = getnode ();
    node->type = NDBMNODE;
```

```
    node->key = xmalloc (key.dsize + 1);
    strncpy (node->key, key.dptr, key.dsize);
    node->key[key.dsize] = '\0';

    node->data = xmalloc (value.dsize + 1);
    strncpy (node->data, value.dptr, value.dsize);
    node->data[value.dsize] = '\0';

    db->modified = 1;
    if (addnode (db->dbm_list, node) == -1)
    {
        error (0, 0, "attempt to insert duplicate key `%s'", node->key);
        freenode (node);
        return 0;
    }
    return 0;
}

static void
mydbm_load_file (fp, list)
    FILE *fp;
    List *list;
{
    char *line = NULL;
    size_t line_size;
    char *value;
    size_t value_allocated;
    char *cp, *vp;
    int cont;
    int line_length;

    value_allocated = 1;
    value = xmalloc (value_allocated);

    cont = 0;
    while ((line_length = getstr (&line, &line_size, fp, '\012', 0)) >= 0)
    {
        if (line_length > 0 && line[line_length - 1] == '\012')
        {
            /* Strip the newline.  */
            --line_length;
            line[line_length] = '\0';
        }
        if (line_length > 0 && line[line_length - 1] == '\015')
        {
            /* If the file (e.g. modules) was written on an NT box, it will
               contain CRLF at the ends of lines.  Strip them (we can't do
               this by opening the file in text mode because we might be
               running on unix).  */
            --line_length;
            line[line_length] = '\0';
        }

        /*
         * Add the line to the value, at the end if this is a continuation
         * line; otherwise at the beginning, but only after any trailing
         * backslash is removed.
         */
        if (!cont)
            value[0] = '\0';

        /*
         * See if the line we read is a continuation line, and strip the
         * backslash if so.
         */
        if (line_length > 0)
            cp = &line[line_length - 1];
        else
            cp = line;
        if (*cp == '\\')
        {
            cont = 1;
            *cp = '\0';
            --line_length;
        }
        else
        {
            cont = 0;
        }
        expand_string (&value,
                       &value_allocated,
                       strlen (value) + line_length + 5);
        strcat (value, line);

        if (value[0] == '#')
            continue;               /* comment line */
        vp = value;
        while (*vp && isspace (*vp))
            vp++;
        if (*vp == '\0')
```

```
            continue;               /* empty line */

        /*
         * If this was not a continuation line, add the entry to the database
         */
        if (!cont)
        {
            Node *p = getnode ();
            char *kp;

            kp = vp;
            while (*vp && !isspace (*vp))
                vp++;
            *vp++ = '\0';           /* NULL terminate the key */
            p->type = NDBMNODE;
            p->key = xstrdup (kp);
            while (*vp && isspace (*vp))
                vp++;               /* skip whitespace to value */
            if (*vp == '\0')
            {
                error (0, 0, "warning: NULL value for key `%s'", p->key);
                freenode (p);
                continue;
            }
            p->data = xstrdup (vp);
            if (addnode (list, p) == -1)
            {
                error (0, 0, "duplicate key found for `%s'", p->key);
                freenode (p);
            }
        }
    }
    if (line_length < 0 && !feof (fp))
        /* FIXME: should give the name of the file.  */
        error (0, errno, "cannot read file in mydbm_load_file");

    free (line);
    free (value);
}

#endif                              /* MY_NDBM */
```

A.40 myndbm.h

```
#ifdef MY_NDBM

#define DBLKSIZ 4096

typedef struct
{
   List *dbm_list;              /* cached database */
   Node *dbm_next;              /* next key to return for nextkey() */

   /* Name of the file to write to if modified is set.  malloc'd.  */
   char *name;

   /* Nonzero if the database has been modified and dbm_close needs to
      write it out to disk.  */
   int modified;
} DBM;

typedef struct
{
   char *dptr;
   int dsize;
} datum;

/*
 * So as not to conflict with other dbm_open, etc., routines that may
 * be included by someone's libc, all of my emulation routines are prefixed
 * by "my" and we define the "standard" ones to be "my" ones here.
 */
#define dbm_open mydbm_open
#define dbm_close mydbm_close
#define dbm_fetch mydbm_fetch
#define dbm_firstkey mydbm_firstkey
#define dbm_nextkey mydbm_nextkey
#define dbm_store mydbm_store
#define  DBM_INSERT  0
#define  DBM_REPLACE 1

DBM *mydbm_open PROTO((char *file, int flags, int mode));
void mydbm_close PROTO((DBM * db));
datum mydbm_fetch PROTO((DBM * db, datum key));
datum mydbm_firstkey PROTO((DBM * db));
datum mydbm_nextkey PROTO((DBM * db));
extern int mydbm_store PROTO ((DBM *, datum, datum, int));

#endif                          /* MY_NDBM */
```

A.41 no_diff.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * No Difference
 *
 * The user file looks modified judging from its time stamp; however it needn't
 * be.  No_difference() finds out whether it is or not. If it is not, it
 * updates the administration.
 *
 * returns 0 if no differences are found and non-zero otherwise
 */

#include "cvs.h"

int
No_Difference (finfo, vers)
    struct file_info *finfo;
    Vers_TS *vers;
{
    Node *p;
    int ret;
    char *ts, *options;
    int retcode = 0;
    char *tocvsPath;

    /* If ts_user is "Is-modified", we can only conclude the files are
       different (since we don't have the file's contents).  */
    if (vers->ts_user != NULL
        && strcmp (vers->ts_user, "Is-modified") == 0)
        return -1;

    if (!vers->srcfile || !vers->srcfile->path)
        return (-1);                /* different since we couldn't tell */

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* If special files are in use, then any mismatch of file metadata
       information also means that the files should be considered different. */
    if (preserve_perms && special_file_mismatch (finfo, vers->vn_user, NULL))
        return 1;
#endif

    if (vers->entdata && vers->entdata->options)
        options = xstrdup (vers->entdata->options);
    else
        options = xstrdup ("");

    tocvsPath = wrap_tocvs_process_file (finfo->file);
    retcode = RCS_cmp_file (vers->srcfile, vers->vn_user, options,
                            tocvsPath == NULL ? finfo->file : tocvsPath);
    if (retcode == 0)
    {
        /* no difference was found, so fix the entries file */
        ts = time_stamp (finfo->file);
        Register (finfo->entries, finfo->file,
                  vers->vn_user ? vers->vn_user : vers->vn_rcs, ts,
                  options, vers->tag, vers->date, (char *) 0, CVSroot_directory, finfo->repository);
#ifdef SERVER_SUPPORT
        if (server_active)
        {
            /* We need to update the entries line on the client side.  */
            server_update_entries
              (finfo->file, finfo->update_dir, finfo->repository, SERVER_UPDATED);
        }
#endif
        free (ts);

        /* update the entdata pointer in the vers_ts structure */
        p = findnode (finfo->entries, finfo->file);
        vers->entdata = (Entnode *) p->data;

        ret = 0;
    }
    else
        ret = 1;                        /* files were really different */

    if (tocvsPath)
    {
        /* Need to call unlink myself because the noexec variable
         * has been set to 1.  */
        if (trace)
            (void) fprintf (stderr, "%c-> unlink (%s)\n",
#ifdef SERVER_SUPPORT
                            (server_active) ? 'S' : ' ',
#else
```

```
                        ' ',
#endif
                        tocvsPath);
        if ( CVS_UNLINK (tocvsPath) < 0)
            error (0, errno, "could not remove %s", tocvsPath);
    }

    free (options);
    return (ret);
}
```

A.42 parseinfo.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"
#include "getline.h"
#include <assert.h>

/*
 * Parse the INFOFILE file for the specified REPOSITORY.  Invoke CALLPROC for
 * the first line in the file that matches the REPOSITORY, or if ALL != 0, any lines
 * matching "ALL", or if no lines match, the last line matching "DEFAULT".
 *
 * Return 0 for success, -1 if there was not an INFOFILE, and >0 for failure.
 */
int
Parse_Info (infofile, repository, callproc, all)
    char *infofile;
    char *repository;
    CALLPROC callproc;
    int all;
{
    int err = 0;
    FILE *fp_info;
    char *infopath;
    char *line = NULL;
    size_t line_allocated = 0;
    char *default_value = NULL;
    char *expanded_value= NULL;
    int callback_done, line_number;
    char *cp, *exp, *value, *srepos;
    const char *regex_err;

    if (CVSroot_original == NULL)
    {
        /* XXX - should be error maybe? */
        error (0, 0, "CVSROOT variable not set");
        return (1);
    }

    /* find the info file and open it */
    infopath = xmalloc (strlen (CVSroot_directory)
                        + strlen (infofile)
                        + sizeof (CVSROOTADM)
                        + 10);
    (void) sprintf (infopath, "%s/%s/%s", CVSroot_directory,
                    CVSROOTADM, infofile);
    fp_info = CVS_FOPEN (infopath, "r");
    if (fp_info == NULL)
    {
        /* If no file, don't do anything special.  */
        if (!existence_error (errno))
            error (0, errno, "cannot open %s", infopath);
        free (infopath);
        return 0;
    }

    /* strip off the CVSROOT if repository was absolute */
    srepos = Short_Repository (repository);

    if (trace)
        (void) fprintf (stderr, " -> ParseInfo(%s, %s, %s)\n",
                        infopath, srepos, all ? "ALL" : "not ALL");

    /* search the info file for lines that match */
    callback_done = line_number = 0;
    while (getline (&line, &line_allocated, fp_info) >= 0)
    {
        line_number++;

        /* skip lines starting with # */
        if (line[0] == '#')
            continue;

        /* skip whitespace at beginning of line */
        for (cp = line; *cp && isspace (*cp); cp++)
            ;

        /* if *cp is null, the whole line was blank */
        if (*cp == '\0')
            continue;

        /* the regular expression is everything up to the first space */
        for (exp = cp; *cp && !isspace (*cp); cp++)
```

```
            ;
        if (*cp != '\0')
            *cp++ = '\0';

        /* skip whitespace up to the start of the matching value */
        while (*cp && isspace (*cp))
            cp++;

        /* no value to match with the regular expression is an error */
        if (*cp == '\0')
        {
            error (0, 0, "syntax error at line %d file %s; ignored",
                   line_number, infofile);
            continue;
        }
        value = cp;

        /* strip the newline off the end of the value */
        if ((cp = strrchr (value, '\n')) != NULL)
            *cp = '\0';

        if (expanded_value != NULL)
            free (expanded_value);
        expanded_value = expand_path (value, infofile, line_number);
        if (!expanded_value)
        {
            continue;
        }

        /*
         * At this point, exp points to the regular expression, and value
         * points to the value to call the callback routine with.  Evaluate
         * the regular expression against srepos and callback with the value
         * if it matches.
         */

        /* save the default value so we have it later if we need it */
        if (strcmp (exp, "DEFAULT") == 0)
        {
            /* Is it OK to silently ignore all but the last DEFAULT
               expression?  */
            if (default_value != NULL)
                free (default_value);
            default_value = xstrdup (expanded_value);
            continue;
        }

        /*
         * For a regular expression of "ALL", do the callback always We may
         * execute lots of ALL callbacks in addition to *one* regular matching
         * callback or default
         */
        if (strcmp (exp, "ALL") == 0)
        {
            if (all)
                err += callproc (repository, expanded_value);
            else
                error(0, 0, "Keyword `ALL' is ignored at line %d in %s file",
                      line_number, infofile);
            continue;
        }

        if (callback_done)
            /* only first matching, plus "ALL"'s */
            continue;

        /* see if the repository matched this regular expression */
        if ((regex_err = re_comp (exp)) != NULL)
        {
            error (0, 0, "bad regular expression at line %d file %s: %s",
                   line_number, infofile, regex_err);
            continue;
        }
        if (re_exec (srepos) == 0)
            continue;                       /* no match */

        /* it did, so do the callback and note that we did one */
        err += callproc (repository, expanded_value);
        callback_done = 1;
    }
    if (ferror (fp_info))
        error (0, errno, "cannot read %s", infopath);
    if (fclose (fp_info) < 0)
        error (0, errno, "cannot close %s", infopath);

    /* if we fell through and didn't callback at all, do the default */
    if (callback_done == 0 && default_value != NULL)
        err += callproc (repository, default_value);

    /* free up space if necessary */
```

```
    if (default_value != NULL)
        free (default_value);
    if (expanded_value != NULL)
        free (expanded_value);
    free (infopath);
    if (line != NULL)
        free (line);

    return (err);
}


/* Parse the CVS config file.  The syntax right now is a bit ad hoc
   but tries to draw on the best or more common features of the other
   *info files and various unix (or non-unix) config file syntaxes.
   Lines starting with # are comments.  Settings are lines of the form
   KEYWORD=VALUE.  There is currently no way to have a multi-line
   VALUE (would be nice if there was, probably).

   CVSROOT is the $CVSROOT directory (CVSroot_directory might not be
   set yet).

   Returns 0 for success, negative value for failure.  Call
   error(0, ...) on errors in addition to the return value.  */
int
parse_config (cvsroot)
    char *cvsroot;
{
    char *infopath;
    FILE *fp_info;
    char *line = NULL;
    size_t line_allocated = 0;
    size_t len;
    char *p;
    /* FIXME-reentrancy: If we do a multi-threaded server, this would need
       to go to the per-connection data structures.  */
    static int parsed = 0;

    /* Authentication code and serve_root might both want to call us.
       Let this happen smoothly.  */
    if (parsed)
        return 0;
    parsed = 1;

    infopath = malloc (strlen (cvsroot)
                       + sizeof (CVSROOTADM_CONFIG)
                       + sizeof (CVSROOTADM)
                       + 10);
    if (infopath == NULL)
    {
        error (0, 0, "out of memory; cannot allocate infopath");
        goto error_return;
    }

    strcpy (infopath, cvsroot);
    strcat (infopath, "/");
    strcat (infopath, CVSROOTADM);
    strcat (infopath, "/");
    strcat (infopath, CVSROOTADM_CONFIG);

    fp_info = CVS_FOPEN (infopath, "r");
    if (fp_info == NULL)
    {
        /* If no file, don't do anything special.  */
        if (!existence_error (errno))
        {
            /* Just a warning message; doesn't affect return
               value, currently at least.  */
            error (0, errno, "cannot open %s", infopath);
        }
        free (infopath);
        return 0;
    }

    while (getline (&line, &line_allocated, fp_info) >= 0)
    {
        /* Skip comments.  */
        if (line[0] == '#')
            continue;

        /* At least for the moment we don't skip whitespace at the start
           of the line.  Too picky?  Maybe.  But being insufficiently
           picky leads to all sorts of confusion, and it is a lot easier
           to start out picky and relax it than the other way around.

           Is there any kind of written standard for the syntax of this
           sort of config file?  Anywhere in POSIX for example (I guess
           makefiles are sort of close)?  Red Hat Linux has a bunch of
           these too (with some GUI tools which edit them)...
```

```
            Along the same lines, we might want a table of keywords,
            with various types (boolean, string, &c), as a mechanism
            for making sure the syntax is consistent.  Any good examples
            to follow there (Apache?)? */

          /* Strip the training newline.  There will be one unless we
             read a partial line without a newline, and then got end of
             file (or error?).  */

          len = strlen (line) - 1;
          if (line[len] == '\n')
            line[len] = '\0';

          /* Skip blank lines.  */
          if (line[0] == '\0')
            continue;

          /* The first '=' separates keyword from value.  */
          p = strchr (line, '=');
          if (p == NULL)
          {
            /* Probably should be printing line number.  */
            error (0, 0, "syntax error in %s: line '%s' is missing '='",
                   infopath, line);
            goto error_return;
          }

          *p++ = '\0';

          if (strcmp (line, "RCSBIN") == 0)
          {
            /* This option used to specify the directory for RCS
               executables.  But since we don't run them any more,
               this is a noop.  Silently ignore it so that a
               repository can work with either new or old CVS.  */
            ;
          }
          else if (strcmp (line, "SystemAuth") == 0)
          {
            if (strcmp (p, "no") == 0)
#ifdef AUTH_SERVER_SUPPORT
              system_auth = 0;
#else
              /* Still parse the syntax but ignore the
                 option.  That way the same config file can
                 be used for local and server.  */
              ;
#endif
            else if (strcmp (p, "yes") == 0)
#ifdef AUTH_SERVER_SUPPORT
              system_auth = 1;
#else
              ;
#endif
            else
            {
              error (0, 0, "unrecognized value '%s' for SystemAuth", p);
              goto error_return;
            }
          }
          else if (strcmp (line, "PreservePermissions") == 0)
          {
            if (strcmp (p, "no") == 0)
              preserve_perms = 0;
            else if (strcmp (p, "yes") == 0)
            {
#ifdef PRESERVE_PERMISSIONS_SUPPORT
              preserve_perms = 1;
#else
              error (0, 0, "\
warning: this CVS does not support PreservePermissions");
#endif
            }
            else
            {
              error (0, 0, "unrecognized value '%s' for PreservePermissions",
                     p);
              goto error_return;
            }
          }
          else if (strcmp (line, "TopLevelAdmin") == 0)
          {
            if (strcmp (p, "no") == 0)
              top_level_admin = 0;
            else if (strcmp (p, "yes") == 0)
              top_level_admin = 1;
            else
            {
              error (0, 0, "unrecognized value '%s' for TopLevelAdmin", p);
              goto error_return;
```

```
            }
        }
        else
        {
            /* We may be dealing with a keyword which was added in a
               subsequent version of CVS.  In that case it is a good idea
               to complain, as (1) the keyword might enable a behavior like
               alternate locking behavior, in which it is dangerous and hard
               to detect if some CVS's have it one way and others have it
               the other way, (2) in general, having us not do what the user
               had in mind when they put in the keyword violates the
               principle of least surprise.  Note that one corollary is
               adding new keywords to your CVSROOT/config file is not
               particularly recommended unless you are planning on using
               the new features.  */
            error (0, 0, "%s: unrecognized keyword '%s'",
                   infopath, line);
            goto error_return;
        }
    }
    if (ferror (fp_info))
    {
        error (0, errno, "cannot read %s", infopath);
        goto error_return;
    }
    if (fclose (fp_info) < 0)
    {
        error (0, errno, "cannot close %s", infopath);
        goto error_return;
    }
    free (infopath);
    if (line != NULL)
        free (line);
    return 0;

error_return:
    if (infopath != NULL)
        free (infopath);
    if (line != NULL)
        free (line);
    return -1;
}
```

A.43 patch.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Patch
 *
 * Create a Larry Wall format "patch" file between a previous release and the
 * current head of a module, or between two releases.  Can specify the
 * release as either a date or a revision number.
 */

#include "cvs.h"
#include "getline.h"

static RETSIGTYPE patch_cleanup PROTO((void));
static Dtype patch_dirproc PROTO ((void *callerdat, char *dir,
                                   char *repos, char *update_dir,
                                   List *entries));
static int patch_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int patch_proc PROTO((int *pargc, char **argv, char *xwhere,
                             char *mwhere, char *mfile, int shorten,
                             int local_specified, char *mname, char *msg));

static int force_tag_match = 1;
static int patch_short = 0;
static int toptwo_diffs = 0;
static int local = 0;
static char *options = NULL;
static char *rev1 = NULL;
static int rev1_validated = 0;
static char *rev2 = NULL;
static int rev2_validated = 0;
static char *date1 = NULL;
static char *date2 = NULL;
static char *tmpfile1 = NULL;
static char *tmpfile2 = NULL;
static char *tmpfile3 = NULL;
static int unidiff = 0;

static const char *const patch_usage[] =
{
    "Usage: %s %s [-flR] [-c|-u] [-s|-t] [-V %%d]\n",
    "    -r rev|-D date [-r rev2 | -D date2] modules. . .\n",
    "\t-f\tForce a head revision match if tag/date not found.\n",
    "\t-l\tLocal directory only, not recursive\n",
    "\t-R\tProcess directories recursively.\n",
    "\t-c\tContext diffs (default)\n",
    "\t-u\tUnidiff format.\n",
    "\t-s\tShort patch - one liner per file.\n",
    "\t-t\tTop two diffs - last change made to the file.\n",
    "\t-D date\tDate.\n",
    "\t-r rev\tRevision - symbolic or numeric.\n",
    "\t-V vers\tUse RCS Version \"vers\" for keyword expansion.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
patch (argc, argv)
    int argc;
    char **argv;
{
    register int i;
    int c;
    int err = 0;
    DBM *db;

    if (argc == -1)
        usage (patch_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+V:k:cuftsQqlRD:r:")) != -1)
    {
        switch (c)
        {
            case 'Q':
            case 'q':
#ifdef SERVER_SUPPORT
                /* The CVS 1.5 client sends these options (in addition to
                   Global_option requests), so we must ignore them.  */
                if (!server_active)
#endif
                    error (1, 0,
                           "-q or -Q must be specified before \"%s\"",
                           command_name);
```

```
            break;
        case 'f':
            force_tag_match = 0;
            break;
        case 'l':
            local = 1;
            break;
        case 'R':
            local = 0;
            break;
        case 't':
            toptwo_diffs = 1;
            break;
        case 's':
            patch_short = 1;
            break;
        case 'D':
            if (rev2 != NULL || date2 != NULL)
                error (1, 0,
                   "no more than two revisions/dates can be specified");
            if (rev1 != NULL || date1 != NULL)
                date2 = Make_Date (optarg);
            else
                date1 = Make_Date (optarg);
            break;
        case 'r':
            if (rev2 != NULL || date2 != NULL)
                error (1, 0,
                   "no more than two revisions/dates can be specified");
            if (rev1 != NULL || date1 != NULL)
                rev2 = optarg;
            else
                rev1 = optarg;
            break;
        case 'k':
            if (options)
                free (options);
            options = RCS_check_kflag (optarg);
            break;
        case 'V':
            /* This option is pretty seriously broken:
               1.  It is not clear what it does (does it change keyword
               expansion behavior?  If so, how?  Or does it have
               something to do with what version of RCS we are using?
               Or the format we write RCS files in?).
               2.  Because both it and -k use the options variable,
               specifying both -V and -k doesn't work.
               3.  At least as of CVS 1.9, it doesn't work (failed
               assertion in RCS_checkout where it asserts that options
               starts with -k).  Few people seem to be complaining.
               In the future (perhaps the near future), I have in mind
               removing it entirely, and updating NEWS and cvs.texinfo,
               but in case it is a good idea to give people more time
               to complain if they would miss it, I'll just add this
               quick and dirty error message for now.  */
            error (1, 0,
                "the -V option is obsolete and should not be used");
#if 0
            if (atoi (optarg) <= 0)
                error (1, 0, "must specify a version number to -V");
            if (options)
                free (options);
            options = xmalloc (strlen (optarg) + 1 + 2); /* for the -V */
            (void) sprintf (options, "-V%s", optarg);
#endif
            break;
        case 'u':
            unidiff = 1;          /* Unidiff */
            break;
        case 'c':                 /* Context diff */
            unidiff = 0;
            break;
        case '?':
        default:
            usage (patch_usage);
            break;
        }
    }
    argc -= optind;
    argv += optind;

    /* Sanity checks */
    if (argc < 1)
        usage (patch_usage);

    if (toptwo_diffs && patch_short)
        error (1, 0, "-t and -s options are mutually exclusive");
    if (toptwo_diffs && (date1 != NULL || date2 != NULL ||
                         rev1 != NULL || rev2 != NULL))
        error (1, 0, "must not specify revisions/dates with -t option!");
```

```
    if (!toptwo_diffs && (date1 == NULL && date2 == NULL &&
                          rev1 == NULL && rev2 == NULL))
        error (1, 0, "must specify at least one revision/date!");
    if (date1 != NULL && date2 != NULL)
        if (RCS_datecmp (date1, date2) >= 0)
            error (1, 0, "second date must come after first date!");

    /* if options is NULL, make it a NULL string */
    if (options == NULL)
        options = xstrdup ("");

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        /* We're the client side.  Fire up the remote server.  */
        start_server ();

        ign_setup ();

        if (local)
            send_arg("-l");
        if (!force_tag_match)
            send_arg("-f");
        if (toptwo_diffs)
            send_arg("-t");
        if (patch_short)
            send_arg("-s");
        if (unidiff)
            send_arg("-u");

        if (rev1)
            option_with_arg ("-r", rev1);
        if (date1)
            client_senddate (date1);
        if (rev2)
            option_with_arg ("-r", rev2);
        if (date2)
            client_senddate (date2);
        if (options[0] != '\0')
            send_arg (options);

        {
            int i;
            for (i = 0; i < argc; ++i)
                send_arg (argv[i]);
        }

        send_to_server ("rdiff\012", 0);
        return get_responses_and_close ();
    }
#endif

    /* clean up if we get a signal */
#ifdef SIGHUP
    (void) SIG_register (SIGHUP, patch_cleanup);
#endif
#ifdef SIGINT
    (void) SIG_register (SIGINT, patch_cleanup);
#endif
#ifdef SIGQUIT
    (void) SIG_register (SIGQUIT, patch_cleanup);
#endif
#ifdef SIGPIPE
    (void) SIG_register (SIGPIPE, patch_cleanup);
#endif
#ifdef SIGTERM
    (void) SIG_register (SIGTERM, patch_cleanup);
#endif

    db = open_module ();
    for (i = 0; i < argc; i++)
        err += do_module (db, argv[i], PATCH, "Patching", patch_proc,
                          (char *) NULL, 0, 0, 0, (char *) NULL);
    close_module (db);
    free (options);
    patch_cleanup ();
    return (err);
}

/*
 * callback proc for doing the real work of patching
 */
/* ARGSUSED */
static int
patch_proc (pargc, argv, xwhere, mwhere, mfile, shorten, local_specified,
            mname, msg)
    int *pargc;
    char **argv;
    char *xwhere;
```

```
    char *mwhere;
    char *mfile;
    int shorten;
    int local_specified;
    char *mname;
    char *msg;
{
    int err = 0;
    int which;
    char *repository;
    char *where;

    repository = xmalloc (strlen (CVSroot_directory) + strlen (argv[0])
                          + (mfile == NULL ? 0 : strlen (mfile)) + 30);
    (void) sprintf (repository, "%s/%s", CVSroot_directory, argv[0]);
    where = xmalloc (strlen (argv[0]) + (mfile == NULL ? 0 : strlen (mfile))
                     + 10);
    (void) strcpy (where, argv[0]);

    /* if mfile isn't null, we need to set up to do only part of the module */
    if (mfile != NULL)
    {
        char *cp;
        char *path;

        /* if the portion of the module is a path, put the dir part on repos */
        if ((cp = strrchr (mfile, '/')) != NULL)
        {
            *cp = '\0';
            (void) strcat (repository, "/");
            (void) strcat (repository, mfile);
            (void) strcat (where, "/");
            (void) strcat (where, mfile);
            mfile = cp + 1;
        }

        /* take care of the rest */
        path = xmalloc (strlen (repository) + strlen (mfile) + 5);
        (void) sprintf (path, "%s/%s", repository, mfile);
        if (isdir (path))
        {
            /* directory means repository gets the dir tacked on */
            (void) strcpy (repository, path);
            (void) strcat (where, "/");
            (void) strcat (where, mfile);
        }
        else
        {
            int i;

            /* a file means muck argv */
            for (i = 1; i < *pargc; i++)
                free (argv[i]);
            argv[1] = xstrdup (mfile);
            (*pargc) = 2;
        }
        free (path);
    }

    /* cd to the starting repository */
    if ( CVS_CHDIR (repository) < 0)
    {
        error (0, errno, "cannot chdir to %s", repository);
        free (repository);
        return (1);
    }
    free (repository);

    if (force_tag_match)
        which = W_REPOS | W_ATTIC;
    else
        which = W_REPOS;

    if (rev1 != NULL && !rev1_validated)
    {
        tag_check_valid (rev1, *pargc - 1, argv + 1, local, 0, NULL);
        rev1_validated = 1;
    }
    if (rev2 != NULL && !rev2_validated)
    {
        tag_check_valid (rev2, *pargc - 1, argv + 1, local, 0, NULL);
        rev2_validated = 1;
    }

    /* start the recursion processor */
    err = start_recursion (patch_fileproc, (FILESDONEPROC) NULL, patch_dirproc,
                           (DIRLEAVEPROC) NULL, NULL,
                           *pargc - 1, argv + 1, local,
                           which, 0, 1, where, 1);
    free (where);
```

```
    return (err);
}

/*
 * Called to examine a particular RCS file, as appropriate with the options
 * that were set above.
 */
/* ARGSUSED */
static int
patch_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    struct utimbuf t;
    char *vers_tag, *vers_head;
    char *rcs = NULL;
    RCSNode *rcsfile;
    FILE *fp1, *fp2, *fp3;
    int ret = 0;
    int isattic = 0;
    int retcode = 0;
    char *file1;
    char *file2;
    char *strippath;
    char *line1, *line2;
    size_t line1_chars_allocated;
    size_t line2_chars_allocated;
    char *cp1, *cp2;
    FILE *fp;
    int line_length;

    line1 = NULL;
    line1_chars_allocated = 0;
    line2 = NULL;
    line2_chars_allocated = 0;

    /* find the parsed rcs file */
    if ((rcsfile = finfo->rcs) == NULL)
    {
        ret = 1;
        goto out2;
    }
    if ((rcsfile->flags & VALID) && (rcsfile->flags & INATTIC))
        isattic = 1;

    rcs = xmalloc (strlen (finfo->file) + sizeof (RCSEXT) + 5);
    (void) sprintf (rcs, "%s%s", finfo->file, RCSEXT);

    /* if vers_head is NULL, may have been removed from the release */
    if (isattic && rev2 == NULL && date2 == NULL)
        vers_head = NULL;
    else
    {
        vers_head = RCS_getversion (rcsfile, rev2, date2, force_tag_match,
                                    (int *) NULL);
        if (vers_head != NULL && RCS_isdead (rcsfile, vers_head))
        {
            free (vers_head);
            vers_head = NULL;
        }
    }

    if (toptwo_diffs)
    {
        if (vers_head == NULL)
        {
            ret = 1;
            goto out2;
        }

        if (!date1)
            date1 = xmalloc (MAXDATELEN);
        *date1 = '\0';
        if (RCS_getrevtime (rcsfile, vers_head, date1, 1) == -1)
        {
            if (!really_quiet)
                error (0, 0, "cannot find date in rcs file %s revision %s",
                       rcs, vers_head);
            ret = 1;
            goto out2;
        }
    }
    vers_tag = RCS_getversion (rcsfile, rev1, date1, force_tag_match,
                               (int *) NULL);
    if (vers_tag != NULL && RCS_isdead (rcsfile, vers_tag))
    {
        free (vers_tag);
        vers_tag = NULL;
    }
```

```
    if (vers_tag == NULL && vers_head == NULL)
    {
        /* Nothing known about specified revs.  */
        ret = 0;
        goto out2;
    }

    if (vers_tag && vers_head && strcmp (vers_head, vers_tag) == 0)
    {
        /* Not changed between releases.  */
        ret = 0;
        goto out2;
    }

    if (patch_short)
    {
        cvs_output ("File ", 0);
        cvs_output (finfo->fullname, 0);
        if (vers_tag == NULL)
        {
            cvs_output (" is new; current revision ", 0);
            cvs_output (vers_head, 0);
            cvs_output ("\n", 1);
        }
        else if (vers_head == NULL)
        {
            cvs_output (" is removed; not included in ", 0);
            if (rev2 != NULL)
            {
                cvs_output ("release tag ", 0);
                cvs_output (rev2, 0);
            }
            else if (date2 != NULL)
            {
                cvs_output ("release date ", 0);
                cvs_output (date2, 0);
            }
            else
                cvs_output ("current release", 0);
            cvs_output ("\n", 1);
        }
        else
        {
            cvs_output (" changed from revision ", 0);
            cvs_output (vers_tag, 0);
            cvs_output (" to ", 0);
            cvs_output (vers_head, 0);
            cvs_output ("\n", 1);
        }
        ret = 0;
        goto out2;
    }

    /* Create 3 empty files.  I'm not really sure there is any advantage
       to doing so now rather than just waiting until later.  */
    tmpfile1 = cvs_temp_name ();
    fp1 = CVS_FOPEN (tmpfile1, "w+");
    if (fp1 == NULL)
    {
        error (0, errno, "cannot create temporary file %s", tmpfile1);
        ret = 1;
        goto out;
    }
    else
        if (fclose (fp1) < 0)
            error (0, errno, "warning: cannot close %s", tmpfile1);
    tmpfile2 = cvs_temp_name ();
    fp2 = CVS_FOPEN (tmpfile2, "w+");
    if (fp2 == NULL)
    {
        error (0, errno, "cannot create temporary file %s", tmpfile2);
        ret = 1;
        goto out;
    }
    else
        if (fclose (fp2) < 0)
            error (0, errno, "warning: cannot close %s", tmpfile2);
    tmpfile3 = cvs_temp_name ();
    fp3 = CVS_FOPEN (tmpfile3, "w+");
    if (fp3 == NULL)
    {
        error (0, errno, "cannot create temporary file %s", tmpfile3);
        ret = 1;
        goto out;
    }
    else
        if (fclose (fp3) < 0)
            error (0, errno, "warning: cannot close %s", tmpfile3);
```

```
    if (vers_tag != NULL)
540 {
        retcode = RCS_checkout (rcsfile, (char *) NULL, vers_tag,
                                rev1, options, tmpfile1,
                                (RCSCHECKOUTPROC) NULL, (void *) NULL);
        if (retcode != 0)
        {
            error (0, 0,
                   "cannot check out revision %s of %s", vers_tag, rcs);
            ret = 1;
            goto out;
550     }
        memset ((char *) &t, 0, sizeof (t));
        if ((t.actime = t.modtime = RCS_getrevtime (rcsfile, vers_tag,
                                                    (char *) 0, 0)) != -1)
        /* I believe this timestamp only affects the dates in our diffs,
           and therefore should be on the server, not the client.  */
        (void) utime (tmpfile1, &t);
    }
    else if (toptwo_diffs)
    {
560     ret = 1;
        goto out;
    }
    if (vers_head != NULL)
    {
        retcode = RCS_checkout (rcsfile, (char *) NULL, vers_head,
                                rev2, options, tmpfile2,
                                (RCSCHECKOUTPROC) NULL, (void *) NULL);
        if (retcode != 0)
        {
570         error (0, 0,
                   "cannot check out revision %s of %s", vers_head, rcs);
            ret = 1;
            goto out;
        }
        if ((t.actime = t.modtime = RCS_getrevtime (rcsfile, vers_head,
                                                    (char *) 0, 0)) != -1)
        /* I believe this timestamp only affects the dates in our diffs,
           and therefore should be on the server, not the client.  */
        (void) utime (tmpfile2, &t);
580 }

    switch (diff_exec (tmpfile1, tmpfile2, unidiff ? "-u" : "-c", tmpfile3))
    {
        case -1:                    /* fork/wait failure */
        error (1, errno, "fork for diff failed on %s", rcs);
            break;
        case 0:                     /* nothing to do */
            break;
        case 1:
590         /*
             * The two revisions are really different, so read the first two
             * lines of the diff output file, and munge them to include more
             * reasonable file names that "patch" will understand.
             */

            /* Output an "Index:" line for patch to use */
            cvs_output ("Index: ", 0);
            cvs_output (finfo->fullname, 0);
            cvs_output ("\n", 1);
600
            fp = open_file (tmpfile3, "r");
            if (getline (&line1, &line1_chars_allocated, fp) < 0 ||
                getline (&line2, &line2_chars_allocated, fp) < 0)
            {
                if (feof (fp))
                    error (0, 0, "\
failed to read diff file header %s for %s: end of file", tmpfile3, rcs);
                else
                    error (0, errno,
610                        "failed to read diff file header %s for %s",
                           tmpfile3, rcs);
                ret = 1;
                if (fclose (fp) < 0)
                    error (0, errno, "error closing %s", tmpfile3);
                goto out;
            }
            if (!unidiff)
            {
                if (strncmp (line1, "*** ", 4) != 0 ||
620                 strncmp (line2, "--- ", 4) != 0 ||
                    (cp1 = strchr (line1, '\t')) == NULL ||
                    (cp2 = strchr (line2, '\t')) == NULL)
                {
                    error (0, 0, "invalid diff header for %s", rcs);
                    ret = 1;
                    if (fclose (fp) < 0)
                        error (0, errno, "error closing %s", tmpfile3);
                    goto out;
```

```
            }
        }
        else
        {
            if (strncmp (line1, "--- ", 4) != 0 ||
                strncmp (line2, "+++ ", 4) != 0 ||
                (cp1 = strchr (line1, '\t')) == NULL ||
                (cp2 = strchr  (line2, '\t')) == NULL)
            {
                error (0, 0, "invalid unidiff header for %s", rcs);
                ret = 1;
                if (fclose (fp) < 0)
                    error (0, errno, "error closing %s", tmpfile3);
                goto out;
            }
        }
        if (CVSroot_directory != NULL)
        {
            strippath = xmalloc (strlen (CVSroot_directory) + 10);
            (void) sprintf (strippath, "%s/", CVSroot_directory);
        }
        else
            strippath = xstrdup (REPOS_STRIP);
        if (strncmp (rcs, strippath, strlen (strippath)) == 0)
            rcs += strlen (strippath);
        free (strippath);
        if (vers_tag != NULL)
        {
            file1 = xmalloc (strlen (finfo->fullname)
                             + strlen (vers_tag)
                             + 10);
            (void) sprintf (file1, "%s:%s", finfo->fullname, vers_tag);
        }
        else
        {
            file1 = xstrdup (DEVNULL);
        }
        file2 = xmalloc (strlen (finfo->fullname)
                         + (vers_head != NULL ? strlen (vers_head) : 10)
                         + 10);
        (void) sprintf (file2, "%s:%s", finfo->fullname,
                        vers_head ? vers_head : "removed");

        /* Note that the string "diff" is specified by POSIX (for -c)
           and is part of the diff output format, not the name of a
           program.  */
        if (unidiff)
        {
            cvs_output ("diff -u ", 0);
            cvs_output (file1, 0);
            cvs_output (" ", 1);
            cvs_output (file2, 0);
            cvs_output ("\n", 1);

            cvs_output ("--- ", 0);
            cvs_output (file1, 0);
            cvs_output (cp1, 0);
            cvs_output ("+++ ", 0);
        }
        else
        {
            cvs_output ("diff -c ", 0);
            cvs_output (file1, 0);
            cvs_output (" ", 1);
            cvs_output (file2, 0);
            cvs_output ("\n", 1);

            cvs_output ("*** ", 0);
            cvs_output (file1, 0);
            cvs_output (cp1, 0);
            cvs_output ("--- ", 0);
        }

        cvs_output (finfo->fullname, 0);
        cvs_output (cp2, 0);

        /* spew the rest of the diff out */
        while ((line_length
                = getline (&line1, &line1_chars_allocated, fp))
               >= 0)
            cvs_output (line1, 0);
        if (line_length < 0 && !feof (fp))
            error (0, errno, "cannot read %s", tmpfile3);

        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", tmpfile3);
        free (file1);
        free (file2);
        break;
    default:
```

```
            error (0, 0, "diff failed for %s", finfo->fullname);
    }
 out:
    if (line1)
        free (line1);
    if (line2)
        free (line2);
    if (CVS_UNLINK (tmpfile1) < 0)
        error (0, errno, "cannot unlink %s", tmpfile1);
    if (CVS_UNLINK (tmpfile2) < 0)
        error (0, errno, "cannot unlink %s", tmpfile2);
    if (CVS_UNLINK (tmpfile3) < 0)
        error (0, errno, "cannot unlink %s", tmpfile3);
    free (tmpfile1);
    free (tmpfile2);
    free (tmpfile3);
    tmpfile1 = tmpfile2 = tmpfile3 = NULL;

 out2:
    if (rcs != NULL)
        free (rcs);
    return (ret);
}

/*
 * Print a warm fuzzy message
 */
/* ARGSUSED */
static Dtype
patch_dirproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    if (!quiet)
        error (0, 0, "Diffing %s", update_dir);
    return (R_PROCESS);
}

/*
 * Clean up temporary files
 */
static RETSIGTYPE
patch_cleanup ()
{
    if (tmpfile1 != NULL)
    {
        (void) unlink_file (tmpfile1);
        free (tmpfile1);
    }
    if (tmpfile2 != NULL)
    {
        (void) unlink_file (tmpfile2);
        free (tmpfile2);
    }
    if (tmpfile3 != NULL)
    {
        (void) unlink_file (tmpfile3);
        free (tmpfile3);
    }
    tmpfile1 = tmpfile2 = tmpfile3 = NULL;
}
```

A.44 rcs.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * The routines contained in this file do all the rcs file parsing and
 * manipulation
 */

#include <assert.h>
#include "cvs.h"
#include "edit.h"
#include "hardlink.h"

int preserve_perms = 0;

/* The RCS -k options, and a set of enums that must match the array.
   These come first so that we can use enum kflag in function
   prototypes.  */
static const char *const kflags[] =
  {"kv", "kvl", "k", "v", "o", "b", (char *) NULL};
enum kflag { KFLAG_KV = 0, KFLAG_KVL, KFLAG_K, KFLAG_V, KFLAG_O, KFLAG_B };

/* A structure we use to buffer the contents of an RCS file.  The
   various fields are only referenced directly by the rcsbuf_*
   functions.  We declare the struct here so that we can allocate it
   on the stack, rather than in memory.  */

struct rcsbuffer
{
    /* Points to the current position in the buffer.  */
    char *ptr;
    /* Points just after the last valid character in the buffer.  */
    char *ptrend;
    /* The file.  */
    FILE *fp;
    /* The name of the file, used for error messages.  */
    const char *filename;
    /* The starting file position of the data in the buffer.  */
    unsigned long pos;
    /* The length of the value.  */
    size_t vlen;
    /* Whether the value contains an '' string.  If so, we can not
       compress whitespace characters.  */
    int at_string;
    /* The number of embedded '' characters in an '' string.  If
       this is non-zero, we must search the string for pairs of ''
       and convert them to a single '*/
    int embedded_at;
};

static RCSNode *RCS_parsercsfile_i PROTO((FILE * fp, const char *rcsfile));
static char *RCS_getdatebranch PROTO((RCSNode * rcs, char *date, char *branch));
static void rcsbuf_open PROTO ((struct rcsbuffer *, FILE *fp,
                                const char *filename, unsigned long pos));
static void rcsbuf_close PROTO ((struct rcsbuffer *));
static int rcsbuf_getkey PROTO ((struct rcsbuffer *, char **keyp,
                                 char **valp));
static int rcsbuf_getrevnum PROTO ((struct rcsbuffer *, char **revp));
static char *rcsbuf_fill PROTO ((struct rcsbuffer *, char *ptr, char **keyp,
                                 char **valp));
static char *rcsbuf_valcopy PROTO ((struct rcsbuffer *, char *val, int polish,
                                    size_t *lenp));
static void rcsbuf_valpolish PROTO ((struct rcsbuffer *, char *val, int polish,
                                     size_t *lenp));
static void rcsbuf_valpolish_internal PROTO ((struct rcsbuffer *, char *to,
                                              const char *from, size_t *lenp));
static unsigned long rcsbuf_ftell PROTO ((struct rcsbuffer *));
static void rcsbuf_get_buffered PROTO ((struct rcsbuffer *, char **datap,
                                        size_t *lenp));
static void rcsbuf_cache PROTO ((RCSNode *, struct rcsbuffer *));
static void rcsbuf_cache_close PROTO ((void));
static void rcsbuf_cache_open PROTO ((RCSNode *, long, FILE **,
                                      struct rcsbuffer *));
static int checkmagic_proc PROTO((Node *p, void *closure));
static void do_branches PROTO((List * list, char *val));
static void do_remote_branches PROTO((List * list, char *val));
static void do_symbols PROTO((List * list, char *val));
static void do_locks PROTO((List * list, char *val));
static void free_rcsnode_contents PROTO((RCSNode *));
static void free_rcsvers_contents PROTO((RCSVers *));
static void rcsvers_delproc PROTO((Node * p));
static char *translate_symtag PROTO((RCSNode *, const char *));
static char *RCS_addbranch PROTO ((RCSNode *, const char *));
static char *truncate_revnum_in_place PROTO ((char *));
static char *truncate_revnum PROTO ((const char *));
static char *printable_date PROTO((const char *));
```

```
static char *escape_keyword_value PROTO ((const char *, int *));
static void expand_keywords PROTO((RCSNode *, RCSVers *, const char *,
                                   const char *, size_t, enum kflag, char *,
                                   size_t, char **, size_t *));
static void cmp_file_buffer PROTO((void *, const char *, size_t));

enum rcs_delta_op {RCS_ANNOTATE, RCS_FETCH};
static void RCS_deltas PROTO ((RCSNode *, FILE *, struct rcsbuffer *, char *,
                               enum rcs_delta_op, char **, size_t *,
                               char **, size_t *));

/* Routines for reading, parsing and writing RCS files. */
static RCSVers *getdelta PROTO ((struct rcsbuffer *, char *, char **,
                                 char **));
static Deltatext *RCS_getdeltatext PROTO ((RCSNode *, FILE *,
                                           struct rcsbuffer *));
static void freedeltatext PROTO ((Deltatext *));

static void RCS_putadmin PROTO ((RCSNode *, FILE *));
static void RCS_putdtree PROTO ((RCSNode *, char *, FILE *));
static void RCS_putdesc PROTO ((RCSNode *, FILE *));
static void putdelta PROTO ((RCSVers *, FILE *));
static int putrcsfield_proc PROTO ((Node *, void *));
static int putsymbol_proc PROTO ((Node *, void *));
static void RCS_copydeltas PROTO ((RCSNode *, FILE *, struct rcsbuffer *,
                                   FILE *, Deltatext *, char *));
static int count_delta_actions PROTO ((Node *, void *));
static void putdeltatext PROTO ((FILE *, Deltatext *));

static FILE *rcs_internal_lockfile PROTO ((char *));
static void rcs_internal_unlockfile PROTO ((FILE *, char *));
static char *rcs_lockfilename PROTO ((char *));
static char* remote_revision_available_locally (struct file_info* finfo,
                                                RCSNode* rcs,
                                                char* branch);

/* The RCS file reading functions are called a lot, and they do some
   string comparisons.  This macro speeds things up a bit by skipping
   the function call when the first characters are different.  It
   evaluates its arguments multiple times.  */
#define STREQ(a, b) ((a)[0] == (b)[0] && strcmp ((a), (b)) == 0)

/*
 * We don't want to use isspace() from the C library because:
 *
 * 1. The definition of "whitespace" in RCS files includes ASCII
 *    backspace, but the C locale doesn't.
 * 2. isspace is an very expensive function call in some implementations
 *    due to the addition of wide character support.
 */
static const char spacetab[] = {
      0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0,  /* 0x00 - 0x0f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x10 - 0x1f */
      1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x20 - 0x2f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x30 - 0x3f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x40 - 0x4f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x50 - 0x5f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x60 - 0x8f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x70 - 0x7f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x80 - 0x8f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0x90 - 0x9f */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0xa0 - 0xaf */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0xb0 - 0xbf */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0xc0 - 0xcf */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0xd0 - 0xdf */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  /* 0xe0 - 0xef */
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0   /* 0xf0 - 0xff */
};

#define whitespace(c) (spacetab[(unsigned char)c] != 0)

/* Parse an rcsfile given a user file name and a repository.  If there is
   an error, we print an error message and return NULL.  If the file
   does not exist, we return NULL without printing anything (I'm not
   sure this allows the caller to do anything reasonable, but it is
   the current behavior).  */
RCSNode *
RCS_parse (file, repos)
    const char *file;
    const char *repos;
{
    RCSNode *rcs;
    FILE *fp;
    RCSNode *retval;
    char *rcsfile;

    /* We're creating a new RCSNode, so there is no hope of finding it
       in the cache.  */
    rcsbuf_cache_close ();
```

```
    rcsfile = xmalloc (strlen (repos) + strlen (file)
                       + sizeof (RCSEXT) + sizeof (CVSATTIC) + 10);
    (void) sprintf (rcsfile, "%s/%s%s", repos, file, RCSEXT);
    if ((fp = CVS_FOPEN (rcsfile, FOPEN_BINARY_READ)) != NULL)
    {
        rcs = RCS_parsercsfile_i(fp, rcsfile);
        if (rcs != NULL)
            rcs->flags |= VALID;

        retval = rcs;
        goto out;
    }
    else if (! existence_error (errno))
    {
        error (0, errno, "cannot open %s", rcsfile);
        retval = NULL;
        goto out;
    }

    (void) sprintf (rcsfile, "%s/%s/%s%s", repos, CVSATTIC, file, RCSEXT);
    if ((fp = CVS_FOPEN (rcsfile, FOPEN_BINARY_READ)) != NULL)
    {
        rcs = RCS_parsercsfile_i(fp, rcsfile);
        if (rcs != NULL)
        {
            rcs->flags |= INATTIC;
            rcs->flags |= VALID;
        }

        retval = rcs;
        goto out;
    }
    else if (! existence_error (errno))
    {
        error (0, errno, "cannot open %s", rcsfile);
        retval = NULL;
        goto out;
    }
#if defined (SERVER_SUPPORT) && !defined (FILENAMES_CASE_INSENSITIVE)
    else if (ign_case)
    {
        int status;
        char *found_path;

        /* The client might be asking for a file which we do have
           (which the client doesn't know about), but for which the
           filename case differs.  We only consider this case if the
           regular CVS_FOPENs fail, because fopen_case is such an
           expensive call.  */
        (void) sprintf (rcsfile, "%s/%s%s", repos, file, RCSEXT);
        status = fopen_case (rcsfile, "rb", &fp, &found_path);
        if (status == 0)
        {
            rcs = RCS_parsercsfile_i (fp, rcsfile);
            if (rcs != NULL)
                rcs->flags |= VALID;

            free (rcs->path);
            rcs->path = found_path;
            retval = rcs;
            goto out;
        }
        else if (! existence_error (status))
        {
            error (0, status, "cannot open %s", rcsfile);
            retval = NULL;
            goto out;
        }

        (void) sprintf (rcsfile, "%s/%s/%s%s", repos, CVSATTIC, file, RCSEXT);
        status = fopen_case (rcsfile, "rb", &fp, &found_path);
        if (status == 0)
        {
            rcs = RCS_parsercsfile_i (fp, rcsfile);
            if (rcs != NULL)
            {
                rcs->flags |= INATTIC;
                rcs->flags |= VALID;
            }

            free (rcs->path);
            rcs->path = found_path;
            retval = rcs;
            goto out;
        }
        else if (! existence_error (status))
        {
            error (0, status, "cannot open %s", rcsfile);
            retval = NULL;
            goto out;
```

```
        }
    }
#endif
    retval = NULL;

 out:
    free (rcsfile);

    return retval;
}

/*
 * Parse a specific rcsfile.
 */
RCSNode *
RCS_parsercsfile (rcsfile)
    char *rcsfile;
{
    FILE *fp;
    RCSNode *rcs;

    /* We're creating a new RCSNode, so there is no hope of finding it
       in the cache.  */
    rcsbuf_cache_close ();

    /* open the rcsfile */
    if ((fp = CVS_FOPEN (rcsfile, FOPEN_BINARY_READ)) == NULL)
    {
        error (0, errno, "Couldn't open rcs file `%s'", rcsfile);
        return (NULL);
    }

    rcs = RCS_parsercsfile_i (fp, rcsfile);

    return (rcs);
}


/*
 */
static RCSNode *
RCS_parsercsfile_i (fp, rcsfile)
    FILE *fp;
    const char *rcsfile;
{
    RCSNode *rdata;
    struct rcsbuffer rcsbuf;
    char *key, *value;

    /* make a node */
    rdata = (RCSNode *) xmalloc (sizeof (RCSNode));
    memset ((char *) rdata, 0, sizeof (RCSNode));
    rdata->refcount = 1;
    rdata->path = xstrdup (rcsfile);

    /* Process HEAD, BRANCH, and EXPAND keywords from the RCS header.

       Most cvs operations on the main branch don't need any more
       information.  Those that do call RCS_reparsercsfile to parse
       the rest of the header and the deltas.  */

    rcsbuf_open (&rcsbuf, fp, rcsfile, 0);

    if (! rcsbuf_getkey (&rcsbuf, &key, &value))
        goto l_error;
    if (STREQ (key, RCSDESC))
        goto l_error;

    if (STREQ (RCSHEAD, key) && value != NULL)
        rdata->head = rcsbuf_valcopy (&rcsbuf, value, 0, (size_t *) NULL);

    if (! rcsbuf_getkey (&rcsbuf, &key, &value))
        goto l_error;
    if (STREQ (key, RCSDESC))
        goto l_error;

    if (STREQ (RCSBRANCH, key) && value != NULL)
    {
        char *cp;

        rdata->branch = rcsbuf_valcopy (&rcsbuf, value, 0, (size_t *) NULL);
        if ((numdots (rdata->branch) & 1) != 0)
        {
            /* turn it into a branch if it's a revision */
            cp = strrchr (rdata->branch, '.');
            *cp = '\0';
        }
    }

    /* Look ahead for expand, stopping when we see desc or a revision
```

```
       number.  */
360    while (1)
       {
           char *cp;

           if (STREQ (RCSEXPAND, key))
           {
               rdata->expand = rcsbuf_valcopy (&rcsbuf, value, 0,
                                               (size_t *) NULL);
               break;
           }
370
           for (cp = key; (isdigit (*cp) || *cp == '.') && *cp != '\0'; cp++)
               /* do nothing */ ;
           if (*cp == '\0')
               break;

           if (STREQ (RCSDESC, key))
               break;

           if (! rcsbuf_getkey (&rcsbuf, &key, &value))
380            break;
       }

       rdata->flags |= PARTIAL;

       rcsbuf_cache (rdata, &rcsbuf);

       return rdata;

   l_error:
390    error (0, 0, "`%s' does not appear to be a valid rcs file",
              rcsfile);
       rcsbuf_close (&rcsbuf);
       freercsnode (&rdata);
       fclose (fp);
       return (NULL);
   }


   /* Do the real work of parsing an RCS file.
400
      On error, die with a fatal error; if it returns at all it was successful.

      If PFP is NULL, close the file when done.  Otherwise, leave it open
      and store the FILE * in *PFP.  */
   void
   RCS_reparsercsfile (rdata, pfp, rcsbufp)
       RCSNode *rdata;
       FILE **pfp;
       struct rcsbuffer *rcsbufp;
410 {
       FILE *fp;
       char *rcsfile;
       struct rcsbuffer rcsbuf;
       Node *q, *kv;
       RCSVers *vnode;
       int gotkey;
       char *cp;
       char *key, *value;

420    assert (rdata != NULL);
       rcsfile = rdata->path;

       rcsbuf_cache_open (rdata, 0, &fp, &rcsbuf);

       /* make a node */
       /* This probably shouldn't be done until later: if a file has an
          empty revision tree (which is permissible), rdata->versions
          should be NULL. -twp */
       rdata->versions = getlist ();
430
       /*
        * process all the special header information, break out when we get to
        * the first revision delta
        */
       gotkey = 0;
       for (;;)
       {
           /* get the next key/value pair */
           if (!gotkey)
440        {
               if (! rcsbuf_getkey (&rcsbuf, &key, &value))
               {
                   error (1, 0, "`%s' does not appear to be a valid rcs file",
                          rcsfile);
               }
           }

           gotkey = 0;
```

```
450         /* Skip head, branch and expand tags; we already have them. */
            if (STREQ (key, RCSHEAD)
                || STREQ (key, RCSBRANCH)
                || STREQ (key, RCSEXPAND))
            {
                continue;
            }

            if (STREQ (key, "access"))
            {
460             if (value != NULL)
                {
                    /* We pass the POLISH parameter as 1 because
                       RCS_addaccess expects nothing but spaces.  FIXME:
                       It would be easy and more efficient to change
                       RCS_addaccess.  */
                    rdata->access = rcsbuf_valcopy (&rcsbuf, value, 1,
                                                    (size_t *) NULL);
                }
                continue;
470         }

            /* We always save lock information, so that we can handle
               -kkvl correctly when checking out a file. */
            if (STREQ (key, "locks"))
            {
                if (value != NULL)
                    rdata->locks_data = rcsbuf_valcopy (&rcsbuf, value, 0,
                                                        (size_t *) NULL);
                if (! rcsbuf_getkey (&rcsbuf, &key, &value))
480             {
                    error (1, 0, "premature end of file reading %s", rcsfile);
                }
                if (STREQ (key, "strict") && value == NULL)
                {
                    rdata->strict_locks = 1;
                }
                else
                    gotkey = 1;
                continue;
490         }

            if (STREQ (RCSSYMBOLS, key))
            {
                if (value != NULL)
                    rdata->symbols_data = rcsbuf_valcopy (&rcsbuf, value, 0,
                                                          (size_t *) NULL);
                continue;
            }

500         /*
             * check key for '."s and digits (probably a rev) if it is a
             * revision or 'desc', we are done with the headers and are down to the
             * revision deltas, so we break out of the loop
             */
            for (cp = key; (isdigit (*cp) || *cp == '.') && *cp != '\0'; cp++)
                /* do nothing */ ;
            /* Note that when comparing with RCSDATE, we are not massaging
               VALUE from the string found in the RCS file.  This is OK
               since we know exactly what to expect.  */
510         if (*cp == '\0' && strncmp (RCSDATE, value, (sizeof RCSDATE) - 1) == 0)
                break;

            if (STREQ (key, RCSDESC))
                break;

            if (STREQ (key, "comment"))
            {
                rdata->comment = rcsbuf_valcopy (&rcsbuf, value, 0,
                                                 (size_t *) NULL);
520             continue;
            }
            if (rdata->other == NULL)
                rdata->other = getlist ();
            kv = getnode ();
            kv->type = RCSFIELD;
            kv->key = xstrdup (key);
            kv->data = rcsbuf_valcopy (&rcsbuf, value, 1, (size_t *) NULL);
            if (addnode (rdata->other, kv) != 0)
            {
530             error (0, 0, "warning: duplicate key `%s' in RCS file `%s'",
                       key, rcsfile);
                freenode (kv);
            }

            /* if we haven't grabbed it yet, we didn't want it */
        }

        /* We got out of the loop, so we have the first part of the first
```

```
        revision delta in KEY (the revision) and VALUE (the date key
        and its value).  This is what getdelta expects to receive.  */

    while ((vnode = getdelta (&rcsbuf, rcsfile, &key, &value)) != NULL)
    {
        /* get the node */
        q = getnode ();
        q->type = RCSVERS;
        q->delproc = rcsvers_delproc;
        q->data = (char *) vnode;
        q->key = vnode->version;

        /* add the nodes to the list */
        if (addnode (rdata->versions, q) != 0)
        {
#if 0
            purify_printf("WARNING: Adding duplicate version: %s (%s)\n",
                    q->key, rcsfile);
            freenode (q);
#endif
        }
    }

    /* Here KEY and VALUE are whatever caused getdelta to return NULL.  */

    if (STREQ (key, RCSDESC))
    {
        if (rdata->desc != NULL)
        {
            error (0, 0,
                   "warning: duplicate key `%s' in RCS file `%s'",
                   key, rcsfile);
            free (rdata->desc);
        }
        /* Don't need to rcsbuf_valcopy `value' because
           getdelta already did that. */
        rdata->desc = xstrdup (value);
    }

    rdata->delta_pos = rcsbuf_ftell (&rcsbuf);

    if (pfp == NULL)
        rcsbuf_cache (rdata, &rcsbuf);
    else
    {
        *pfp = fp;
        *rcsbufp = rcsbuf;
    }
    rdata->flags &= ~PARTIAL;
}

/*
 * Fully parse the RCS file.  Store all keyword/value pairs, fetch the
 * log messages for each revision, and fetch add and delete counts for
 * each revision (we could fetch the entire text for each revision,
 * but the only caller, log_fileproc, doesn't need that information,
 * so we don't waste the memory required to store it).  The add and
 * delete counts are stored on the OTHER field of the RCSVERSNODE
 * structure, under the names ";add" and ";delete", so that we don't
 * waste the memory space of extra fields in RCSVERSNODE for code
 * which doesn't need this information.
 */

void
RCS_fully_parse (rcs)
    RCSNode *rcs;
{
    FILE *fp;
    struct rcsbuffer rcsbuf;

    RCS_reparsercsfile (rcs, &fp, &rcsbuf);

    while (1)
    {
        char *key, *value;
        Node *vers;
        RCSVers *vnode;

        /* Rather than try to keep track of how much information we
           have read, just read to the end of the file.  */
        if (! rcsbuf_getrevnum (&rcsbuf, &key))
            break;

        vers = findnode (rcs->versions, key);
        if (vers == NULL)
            error (1, 0,
                   "mismatch in rcs file %s between deltas and deltatexts",
                   rcs->path);

        vnode = (RCSVers *) vers->data;
```

```
630         while (rcsbuf_getkey (&rcsbuf, &key, &value))
            {
              if (! STREQ (key, "text"))
              {
                Node *kv;

                if (vnode->other == NULL)
                  vnode->other = getlist ();
                kv = getnode ();
                kv->type = RCSFIELD;
640             kv->key = xstrdup (key);
                kv->data = rcsbuf_valcopy (&rcsbuf, value, 1, (size_t *) NULL);
                if (addnode (vnode->other, kv) != 0)
                {
                  error (0, 0,
                         "\
warning: duplicate key `%s' in version `%s' of RCS file `%s'",
                         key, vnode->version, rcs->path);
                  freenode (kv);
                }
650
                continue;
              }

              if (! STREQ (vnode->version, rcs->head))
              {
                unsigned long add, del;
                char buf[50];
                Node *kv;

660             /* This is a change text.  Store the add and delete
                   counts.  */
                add = 0;
                del = 0;
                if (value != NULL)
                {
                  size_t vallen;
                  const char *cp;

                  rcsbuf_valpolish (&rcsbuf, value, 0, &vallen);
670               cp = value;
                  while (cp < value + vallen)
                  {
                    char op;
                    unsigned long count;

                    op = *cp++;
                    if (op != 'a' && op  != 'd')
                      error (1, 0, "unrecognized operation '%c' in %s",
                             op, rcs->path);
680                 (void) strtoul (cp, (char **) &cp, 10);
                    if (*cp++ != ' ')
                      error (1, 0, "space expected in %s",
                             rcs->path);
                    count = strtoul (cp, (char **) &cp, 10);
                    if (*cp++ != '\012')
                      error (1, 0, "linefeed expected in %s",
                             rcs->path);

                    if (op == 'd')
690                   del += count;
                    else
                    {
                      add += count;
                      while (count != 0)
                      {
                        if (*cp == '\012')
                          --count;
                        else if (cp == value + vallen)
                        {
700                       if (count != 1)
                            error (1, 0, "\
invalid rcs file %s: premature end of value",
                                   rcs->path);
                          else
                            break;
                        }
                        ++cp;
                      }
                    }
                  }
710             }

                sprintf (buf, "%lu", add);
                kv = getnode ();
                kv->type = RCSFIELD;
                kv->key = xstrdup (";add");
                kv->data = xstrdup (buf);
                if (addnode (vnode->other, kv) != 0)
```

```
            {
720             error (0, 0,
                       "\
warning: duplicate key `%s' in version `%s' of RCS file `%s'",
                       key, vnode->version, rcs->path);
                freenode (kv);
            }

            sprintf (buf, "%lu", del);
            kv = getnode ();
            kv->type = RCSFIELD;
730         kv->key = xstrdup (";delete");
            kv->data = xstrdup (buf);
            if (addnode (vnode->other, kv) != 0)
            {
                error (0, 0,
                       "\
warning: duplicate key `%s' in version `%s' of RCS file `%s'",
                       key, vnode->version, rcs->path);
                freenode (kv);
            }
740       }

          /* We have found the "text" key which ends the data for
             this revision.  Break out of the loop and go on to the
             next revision.  */
          break;
        }
    }

    rcsbuf_cache (rcs, &rcsbuf);
750 }

    /*
     * freercsnode - free up the info for an RCSNode
     */
    void
    freercsnode (rnodep)
        RCSNode **rnodep;
    {
        if (rnodep == NULL || *rnodep == NULL)
760         return;

        ((*rnodep)->refcount)--;
        if ((*rnodep)->refcount != 0)
        {
            *rnodep = (RCSNode *) NULL;
            return;
        }
        free ((*rnodep)->path);
        if ((*rnodep)->head != (char *) NULL)
770         free ((*rnodep)->head);
        if ((*rnodep)->branch != (char *) NULL)
            free ((*rnodep)->branch);
        free_rcsnode_contents (*rnodep);
        free ((char *) *rnodep);
        *rnodep = (RCSNode *) NULL;
    }

    /*
     * free_rcsnode_contents - free up the contents of an RCSNode without
780  * freeing the node itself, or the file name, or the head, or the
     * path.  This returns the RCSNode to the state it is in immediately
     * after a call to RCS_parse.
     */
    static void
    free_rcsnode_contents (rnode)
        RCSNode *rnode;
    {
        dellist (&rnode->versions);
        if (rnode->symbols != (List *) NULL)
790         dellist (&rnode->symbols);
        if (rnode->symbols_data != (char *) NULL)
            free (rnode->symbols_data);
        if (rnode->expand != NULL)
            free (rnode->expand);
        if (rnode->other != (List *) NULL)
            dellist (&rnode->other);
        if (rnode->access != NULL)
            free (rnode->access);
        if (rnode->locks_data != NULL)
800         free (rnode->locks_data);
        if (rnode->locks != (List *) NULL)
            dellist (&rnode->locks);
        if (rnode->comment != NULL)
            free (rnode->comment);
        if (rnode->desc != NULL)
            free (rnode->desc);
    }
```

```
/* free_rcsvers_contents - free up the contents of an RCSVers node,
   but also free the pointer to the node itself. */
/* Note: The 'hardlinks' list is *not* freed, since it is merely a
   pointer into the 'hardlist' structure (defined in hardlink.c), and
   that structure is freed elsewhere in the program. */

static void
free_rcsvers_contents (rnode)
    RCSVers *rnode;
{
    if (rnode->branches != (List *) NULL)
        dellist (&rnode->branches);
    if (rnode->date != (char *) NULL)
        free (rnode->date);
    if (rnode->next != (char *) NULL)
        free (rnode->next);
    if (rnode->author != (char *) NULL)
        free (rnode->author);
    if (rnode->state != (char *) NULL)
        free (rnode->state);
    if (rnode->other != (List *) NULL)
        dellist (&rnode->other);
    if (rnode->other_delta != NULL)
        dellist (&rnode->other_delta);
    if (rnode->text != NULL)
        freedeltatext (rnode->text);
    free ((char *) rnode);
}

/*
 * rcsvers_delproc - free up an RCSVers type node
 */
static void
rcsvers_delproc (p)
    Node *p;
{
    free_rcsvers_contents ((RCSVers *) p->data);
}

/* These functions retrieve keys and values from an RCS file using a
   buffer.  We use this somewhat complex approach because it turns out
   that for many common operations, CVS spends most of its time
   reading keys, so it's worth doing some fairly hairy optimization.  */

/* The number of bytes we try to read each time we need more data.  */

#define RCSBUF_BUFSIZE (8192)

/* The buffer we use to store data.  This grows as needed.  */

static char *rcsbuf_buffer = NULL;
static size_t rcsbuf_buffer_size = 0;

/* Whether rcsbuf_buffer is in use.  This is used as a sanity check.  */

static int rcsbuf_inuse;

/* Set up to start gathering keys and values from an RCS file.  This
   initializes RCSBUF.  */

static void
rcsbuf_open (rcsbuf, fp, filename, pos)
    struct rcsbuffer *rcsbuf;
    FILE *fp;
    const char *filename;
    unsigned long pos;
{
    if (rcsbuf_inuse)
        error (1, 0, "rcsbuf_open: internal error");
    rcsbuf_inuse = 1;

    if (rcsbuf_buffer_size < RCSBUF_BUFSIZE)
        expand_string (&rcsbuf_buffer, &rcsbuf_buffer_size, RCSBUF_BUFSIZE);

    rcsbuf->ptr = rcsbuf_buffer;
    rcsbuf->ptrend = rcsbuf_buffer;
    rcsbuf->fp = fp;
    rcsbuf->filename = filename;
    rcsbuf->pos = pos;
    rcsbuf->vlen = 0;
    rcsbuf->at_string = 0;
    rcsbuf->embedded_at = 0;
}

/* Stop gathering keys from an RCS file.  */

static void
rcsbuf_close (rcsbuf)
    struct rcsbuffer *rcsbuf;
{
```

```
    if (! rcsbuf_inuse)
900     error (1, 0, "rcsbuf_close: internal error");
    rcsbuf_inuse = 0;
}

/* Read a key/value pair from an RCS file.  This sets *KEYP to point
   to the key, and *VALUEP to point to the value.  A missing or empty
   value is indicated by setting *VALUEP to NULL.

   This function returns 1 on success, or 0 on EOF.  If there is an
   error reading the file, or an EOF in an unexpected location, it
910 gives a fatal error.

   This sets *KEYP and *VALUEP to point to storage managed by
   rcsbuf_getkey.  Moreover, *VALUEP has not been massaged from the
   RCS format: it may contain embedded whitespace and embedded ''
   characters.  Call rcsbuf_valcopy or rcsbuf_valpolish to do
   appropriate massaging.  */

static int
rcsbuf_getkey (rcsbuf, keyp, valp)
920 struct rcsbuffer *rcsbuf;
    char **keyp;
    char **valp;
{
    register const char * const my_spacetab = spacetab;
    register char *ptr, *ptrend;
    char c;

#define my_whitespace(c)  (my_spacetab[(unsigned char)c] != 0)

930 rcsbuf->vlen = 0;
    rcsbuf->at_string = 0;
    rcsbuf->embedded_at = 0;

    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;

    /* Sanity check.  */
    if (ptr < rcsbuf_buffer || ptr > rcsbuf_buffer + rcsbuf_buffer_size)
        abort ();
940
    /* If the pointer is more than RCSBUF_BUFSIZE bytes into the
       buffer, move back to the start of the buffer.  This keeps the
       buffer from growing indefinitely.  */
    if (ptr - rcsbuf_buffer >= RCSBUF_BUFSIZE)
    {
        int len;

        len = ptrend - ptr;

950     /* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes
*/
        if (len > RCSBUF_BUFSIZE)
            abort ();

        /* Update the POS field, which holds the file offset of the
           first byte in the RCSBUF_BUFFER buffer.  */
        rcsbuf->pos += ptr - rcsbuf_buffer;

        memcpy (rcsbuf_buffer, ptr, len);
960     ptr = rcsbuf_buffer;
        ptrend = ptr + len;
        rcsbuf->ptrend = ptrend;
    }

    /* Skip leading whitespace.  */

    while (1)
    {
        if (ptr >= ptrend)
970     {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
            if (ptr == NULL)
                return 0;
            ptrend = rcsbuf->ptrend;
        }

        c = *ptr;
        if (! my_whitespace (c))
            break;
980
        ++ptr;
    }

    /* We've found the start of the key.  */

    *keyp = ptr;

    if (c != ';')
```

```
{
    while (1)
    {
        ++ptr;
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, keyp, (char **) NULL);
            if (ptr == NULL)
                error (1, 0, "EOF in key in RCS file %s",
                       rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
        }
        c = *ptr;
        if (c == ';' || my_whitespace (c))
            break;
    }
}

/* Here *KEYP points to the key in the buffer, C is the character
   we found at the of the key, and PTR points to the location in
   the buffer where we found C.  We must set *PTR to \0 in order
   to terminate the key.  If the key ended with ';', then there is
   no value.  */

*ptr = '\0';
++ptr;

if (c == ';')
{
    *valp = NULL;
    rcsbuf->ptr = ptr;
    return 1;
}

/* C must be whitespace.  Skip whitespace between the key and the
   value.  If we find ';' now, there is no value.  */

while (1)
{
    if (ptr >= ptrend)
    {
        ptr = rcsbuf_fill (rcsbuf, ptr, keyp, (char **) NULL);
        if (ptr == NULL)
            error (1, 0, "EOF while looking for value in RCS file %s",
                   rcsbuf->filename);
        ptrend = rcsbuf->ptrend;
    }
    c = *ptr;
    if (c == ';')
    {
        *valp = NULL;
        rcsbuf->ptr = ptr + 1;
        return 1;
    }
    if (! my_whitespace (c))
        break;
    ++ptr;
}

/* Now PTR points to the start of the value, and C is the first
   character of the value.  */

if (c != '@')
    *valp = ptr;
else
{
    char *pat;
    size_t vlen;

    /* Optimize the common case of a value composed of a single
       '*/

    rcsbuf->at_string = 1;

    ++ptr;

    *valp = ptr;

    while (1)
    {
        while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)
        {
            /* Note that we pass PTREND as the PTR value to
               rcsbuf_fill, so that we will wind up setting PTR to
               the location corresponding to the old PTREND, so
               that we don't search the same bytes again.  */
            ptr = rcsbuf_fill (rcsbuf, ptrend, keyp, valp);
            if (ptr == NULL)
                error (1, 0,
                       "EOF while looking for end of string in RCS file %s",
```

```
                    rcsbuf->filename);
1080            ptrend = rcsbuf->ptrend;
            }

            /* Handle the special case of an '' right at the end of
               the known bytes.  */
            if (pat + 1 >= ptrend)
            {
               /* Note that we pass PAT, not PTR, here.  */
               pat = rcsbuf_fill (rcsbuf, pat, keyp, valp);
               if (pat == NULL)
1090           {
                  /* EOF here is OK; it just means that the last
                     character of the file was an '' terminating a
                     value for a key type which does not require a
                     trailing ';'.  */
                  pat = rcsbuf->ptrend - 1;

               }
               ptrend = rcsbuf->ptrend;

1100           /* Note that the value of PTR is bogus here.  This is
                  OK, because we don't use it.  */
            }

            if (pat + 1 >= ptrend || pat[1] != '@')
               break;

            /* We found an '@'.  */
            ++rcsbuf->embedded_at;
            ptr = pat + 2;
1110        }

        /* Here PAT points to the final '@'.  */

        *pat = '\0';

        vlen = pat - *valp;
        if (vlen == 0)
            *valp = NULL;
        rcsbuf->vlen = vlen;
1120
        ptr = pat + 1;
    }

    /* Certain keywords only have a '' string.  If there is no ''
       string, then the old getrcskey function assumed that they had
       no value, and we do the same.  */

    {
        char *k;
1130
        k = *keyp;
        if (STREQ (k, RCSDESC)
            || STREQ (k, "text")
            || STREQ (k, "log"))
        {
            if (c != '@')
                *valp = NULL;
            rcsbuf->ptr = ptr;
            return 1;
1140        }
    }

    /* If we've already gathered a '' string, try to skip whitespace
       and find a '@'.  */
    if (c == '@')
    {
        while (1)
        {
            char n;
1150
            if (ptr >= ptrend)
            {
                ptr = rcsbuf_fill (rcsbuf, ptr, keyp, valp);
                if (ptr == NULL)
                    error (1, 0, "EOF in value in RCS file %s",
                           rcsbuf->filename);
                ptrend = rcsbuf->ptrend;
            }
            n = *ptr;
1160        if (n == ';')
            {
                /* We're done.  We already set everything up for this
                   case above.  */
                rcsbuf->ptr = ptr + 1;
                return 1;
            }
            if (! my_whitespace (n))
                break;
```

```
            ++ptr;
        }

        /* The value extends past the '@' string.  We need to undo the
           closing of the '@' done in the default case above.  This
           case never happens in a plain RCS file, but it can happen
           if user defined phrases are used.  */
        if (rcsbuf->vlen != 0)
            (*valp)[rcsbuf->vlen] = '@';
        else
            *valp = ptr;
    }

    /* Here we have a value which is not a simple '@' string.  We need
       to gather up everything until the next ';', including any '@'
       strings.  *VALP points to the start of the value.  If
       RCSBUF->VLEN is not zero, then we have already read an '@'
       string, and PTR points to the data following the '@' string.
       Otherwise, PTR points to the start of the value.  */

    while (1)
    {
        char *start, *psemi, *pat;

        /* Find the ';' which must end the value.  */
        start = ptr;
        while ((psemi = memchr (ptr, ';', ptrend - ptr)) == NULL)
        {
            int slen;

            /* Note that we pass PTREND as the PTR value to
               rcsbuf_fill, so that we will wind up setting PTR to the
               location corresponding to the old PTREND, so that we
               don't search the same bytes again.  */
            slen = start - *valp;
            ptr = rcsbuf_fill (rcsbuf, ptrend, keyp, valp);
            if (ptr == NULL)
                error (1, 0, "EOF in value in RCS file %s", rcsbuf->filename);
            start = *valp + slen;
            ptrend = rcsbuf->ptrend;
        }

        /* See if there are any '@'.  */
        pat = memchr (start, '@', psemi - start);

        if (pat == NULL)
        {
            size_t vlen;

            /* We're done with the value.  Trim any trailing
               whitespace.  */

            rcsbuf->ptr = psemi + 1;

            start = *valp;
            while (psemi > start && my_whitespace (psemi[-1]))
                --psemi;
            *psemi = '\0';

            vlen = psemi - start;
            if (vlen == 0)
                *valp = NULL;
            rcsbuf->vlen = vlen;

            return 1;
        }

        /* We found an '@' string in the value.  We set
           RCSBUF->AT_STRING, which means that we won't be able to
           compress whitespace correctly for this type of value.
           Since this type of value never arises in a normal RCS file,
           this should not be a big deal.  It means that if anybody
           adds a phrase which can have both an '@' string and regular
           text, they will have to handle whitespace compression
    */

        rcsbuf->at_string = 1;

        *pat = '@';

        ptr = pat + 1;

        while (1)
        {
            while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)
            {
                /* Note that we pass PTREND as the PTR value to
                   rcsbuff_fill, so that we will wind up setting PTR
                   to the location corresponding to the old PTREND, so
                   that we don't search the same bytes again.  */
```

```
            ptr = rcsbuf_fill (rcsbuf, ptrend, keyp, valp);
1260        if (ptr == NULL)
              error (1, 0,
                     "EOF while looking for end of string in RCS file %s",
                     rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
          }

          /* Handle the special case of an '' right at the end of
             the known bytes.  */
          if (pat + 1 >= ptrend)
1270      {
            ptr = rcsbuf_fill (rcsbuf, ptr, keyp, valp);
            if (ptr == NULL)
              error (1, 0, "EOF in value in RCS file %s",
                     rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
          }

          if (pat[1] != ')
            break;
1280
          /* We found an '*/
          ++rcsbuf->embedded_at;
          ptr = pat + 2;
        }

        /* Here PAT points to the final '*/

        *pat = ' ';

1290    ptr = pat + 1;
      }

    #undef my_whitespace
    }

    /* TODO: Eliminate redundant code in rcsbuf_getkey, rcsbuf_getid,
       rcsbuf_getstring, rcsbuf_getword.  These last three functions were
       all created by hacking monstrous swaths of code from rcsbuf_getkey,
       and some engineering would make the code easier to read and
1300   maintain.

       Note that the extreme hair in rcsbuf_getkey is because profiling
       statistics show that it was worth it.

       We probably could be processing "hardlinks" by first calling
       rcsbuf_getkey, and breaking up the value afterwards; the code to
       break it up would not need to be hacked for speed.  This would
       remove the need for rcsbuf_getword, rcsbuf_getid, and
       rcsbuf_getstring, as the other calls are easy to remove.  */
1310
    /* Read an 'id' (in the sense of rcsfile(5)) from RCSBUF, and store in
       IDP. */

    static int
    rcsbuf_getid (rcsbuf, idp)
        struct rcsbuffer *rcsbuf;
        char **idp;
    {
        register const char * const my_spacetab = spacetab;
1320    register char *ptr, *ptrend;
        char c;

    #define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)

        rcsbuf->vlen = 0;
        rcsbuf->at_string = 0;
        rcsbuf->embedded_at = 0;

        ptr = rcsbuf->ptr;
1330    ptrend = rcsbuf->ptrend;

        /* Sanity check.  */
        if (ptr < rcsbuf_buffer || ptr > rcsbuf_buffer + rcsbuf_buffer_size)
            abort ();

        /* If the pointer is more than RCSBUF_BUFSIZE bytes into the
           buffer, move back to the start of the buffer.  This keeps the
           buffer from growing indefinitely.  */
        if (ptr - rcsbuf_buffer >= RCSBUF_BUFSIZE)
1340    {
            int len;

            len = ptrend - ptr;

            /* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes
               at a time, so we can't have more bytes than that past PTR.  */
            if (len > RCSBUF_BUFSIZE)
                abort ();
```

```
1350        /* Update the POS field, which holds the file offset of the
               first byte in the RCSBUF_BUFFER buffer.  */
            rcsbuf->pos += ptr - rcsbuf_buffer;

            memcpy (rcsbuf_buffer, ptr, len);
            ptr = rcsbuf_buffer;
            ptrend = ptr + len;
            rcsbuf->ptrend = ptrend;
        }

1360    /* Skip leading whitespace.  */

        while (1)
        {
            if (ptr >= ptrend)
            {
                ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
                if (ptr == NULL)
                    return 0;
                ptrend = rcsbuf->ptrend;
1370        }

            c = *ptr;
            if (! my_whitespace (c))
                break;

            ++ptr;
        }

        /* We've found the start of the key.  */
1380
        *idp = ptr;

        if (c != ';')
        {
            while (1)
            {
                ++ptr;
                if (ptr >= ptrend)
                {
1390                ptr = rcsbuf_fill (rcsbuf, ptr, idp, (char **) NULL);
                    if (ptr == NULL)
                        error (1, 0, "EOF in key in RCS file %s",
                               rcsbuf->filename);
                    ptrend = rcsbuf->ptrend;
                }
                c = *ptr;
                if (c == ';' || my_whitespace (c))
                    break;
            }
1400    }

        /* Here *IDP points to the id in the buffer, C is the character
           we found at the end of the key, and PTR points to the location in
           the buffer where we found C.  We may not set *PTR to \0, because
           it may overwrite a terminating semicolon.  The calling function
           must copy and terminate the id on its own. */

        rcsbuf->ptr = ptr;
        return 1;
1410
    #undef my_whitespace
    }

    /* Read an RCS */

    static int
    rcsbuf_getstring (rcsbuf, strp)
        struct rcsbuffer *rcsbuf;
        char **strp;
1420 {
        register const char * const my_spacetab = spacetab;
        register char *ptr, *ptrend;
        char *pat;
        size_t vlen;
        char c;

    #define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)

        rcsbuf->vlen = 0;
1430    rcsbuf->at_string = 0;
        rcsbuf->embedded_at = 0;

        ptr = rcsbuf->ptr;
        ptrend = rcsbuf->ptrend;

        /* Sanity check.  */
        if (ptr < rcsbuf_buffer || ptr > rcsbuf_buffer + rcsbuf_buffer_size)
            abort ();
```

```
1440    /* If the pointer is more than RCSBUF_BUFSIZE bytes into the
           buffer, move back to the start of the buffer.  This keeps the
           buffer from growing indefinitely.  */
        if (ptr - rcsbuf_buffer >= RCSBUF_BUFSIZE)
        {
            int len;

            len = ptrend - ptr;

            /* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes
1450           at a time, so we can't have more bytes than that past PTR.  */
            if (len > RCSBUF_BUFSIZE)
                abort ();

            /* Update the POS field, which holds the file offset of the
               first byte in the RCSBUF_BUFFER buffer.  */
            rcsbuf->pos += ptr - rcsbuf_buffer;

            memcpy (rcsbuf_buffer, ptr, len);
            ptr = rcsbuf_buffer;
1460        ptrend = ptr + len;
            rcsbuf->ptrend = ptrend;
        }

        /* Skip leading whitespace.  */

        while (1)
        {
            if (ptr >= ptrend)
            {
1470            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
                if (ptr == NULL)
                    error (1, 0, "unexpected end of file reading %s",
                           rcsbuf->filename);
                ptrend = rcsbuf->ptrend;
            }

            c = *ptr;
            if (! my_whitespace (c))
                break;
1480
            ++ptr;
        }

        /* PTR should now point to the start of a string. */
        if (c != '@')
            error (1, 0, "expected @-string at `%c' in %s", c, rcsbuf->filename);

        /* Optimize the common case of a value composed of a single
           '*/
1490
        rcsbuf->at_string = 1;

        ++ptr;

        *strp = ptr;

        while (1)
        {
            while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)
1500        {
                /* Note that we pass PTREND as the PTR value to
                   rcsbuf_fill, so that we will wind up setting PTR to
                   the location corresponding to the old PTREND, so
                   that we don't search the same bytes again.  */
                ptr = rcsbuf_fill (rcsbuf, ptrend, NULL, strp);
                if (ptr == NULL)
                    error (1, 0,
                           "EOF while looking for end of string in RCS file %s",
                           rcsbuf->filename);
1510            ptrend = rcsbuf->ptrend;
            }

            /* Handle the special case of an '' right at the end of
              the known bytes.  */
            if (pat + 1 >= ptrend)
            {
                /* Note that we pass PAT, not PTR, here.  */
                pat = rcsbuf_fill (rcsbuf, pat, NULL, strp);
                if (pat == NULL)
1520            {
                    /* EOF here is OK; it just means that the last
                       character of the file was an ' terminating a
                       value for a key type which does not require a
                       trailing ';'.  */
                    pat = rcsbuf->ptrend - 1;

                }
                ptrend = rcsbuf->ptrend;
```

```
            /* Note that the value of PTR is bogus here.  This is
               OK, because we don't use it.  */
          }

          if (pat + 1 >= ptrend || pat[1] != '@')
            break;

          /* We found an '*/
          ++rcsbuf->embedded_at;
          ptr = pat + 2;
        }

      /* Here PAT points to the final '*/

      *pat = '\0';

      vlen = pat - *strp;
      if (vlen == 0)
        *strp = NULL;
      rcsbuf->vlen = vlen;
      rcsbuf->ptr = pat + 1;

      return 1;

    #undef my_whitespace
    }

    /* Read an RCS 'word', in the sense of rcsfile(5) (an id, a num, a
       -delimited string, or ':'). Store the result in WORDP. If a
       ';'*/

    static int
    rcsbuf_getword (rcsbuf, wordp)
        struct rcsbuffer *rcsbuf;
        char **wordp;
    {
        register const char * const my_spacetab = spacetab;
        register char *ptr, *ptrend;
        char c;

    #define my_whitespace(c) (my_spacetab[(unsigned char)c] != 0)

        rcsbuf->vlen = 0;
        rcsbuf->at_string = 0;
        rcsbuf->embedded_at = 0;

        ptr = rcsbuf->ptr;
        ptrend = rcsbuf->ptrend;

        /* Sanity check.  */
        if (ptr < rcsbuf_buffer || ptr > rcsbuf_buffer + rcsbuf_buffer_size)
          abort ();

        /* If the pointer is more than RCSBUF_BUFSIZE bytes into the
           buffer, move back to the start of the buffer.  This keeps the
           buffer from growing indefinitely.  */
        if (ptr - rcsbuf_buffer >= RCSBUF_BUFSIZE)
        {
            int len;

            len = ptrend - ptr;

            /* Sanity check: we don't read more than RCSBUF_BUFSIZE bytes
               at a time, so we can't have more bytes than that past PTR.  */
            if (len > RCSBUF_BUFSIZE)
              abort ();

            /* Update the POS field, which holds the file offset of the
               first byte in the RCSBUF_BUFFER buffer.  */
            rcsbuf->pos += ptr - rcsbuf_buffer;

            memcpy (rcsbuf_buffer, ptr, len);
            ptr = rcsbuf_buffer;
            ptrend = ptr + len;
            rcsbuf->ptrend = ptrend;
        }

        /* Skip leading whitespace.  */

        while (1)
        {
            if (ptr >= ptrend)
            {
                ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
                if (ptr == NULL)
                    error (1, 0, "unexpected end of file reading %s",
                           rcsbuf->filename);
                ptrend = rcsbuf->ptrend;
            }
```

```
1620        c = *ptr;
            if (! my_whitespace (c))
              break;

            ++ptr;
          }

          /* If we have reached ';', there is no value. */
          if (c == ';')
          {
1630        *wordp = NULL;
            *ptr++ = '\0';
            rcsbuf->ptr = ptr;
            rcsbuf->vlen = 0;
            return 1;
          }

          /* PTR now points to the start of a value.  Find out whether it is
             a num, an id, a string or a colon. */
          if (c == ':')
1640      {
            *wordp = ptr++;
            rcsbuf->ptr = ptr;
            rcsbuf->vlen = 1;
            return 1;
          }

          if (c == '@')
          {
            char *pat;
1650        size_t vlen;

            /* Optimize the common case of a value composed of a single
               '*/

            rcsbuf->at_string = 1;

            ++ptr;

            *wordp = ptr;
1660
            while (1)
            {
              while ((pat = memchr (ptr, '@', ptrend - ptr)) == NULL)
              {
                /* Note that we pass PTREND as the PTR value to
                   rcsbuf_fill, so that we will wind up setting PTR to
                   the location corresponding to the old PTREND, so
                   that we don't search the same bytes again.  */
                ptr = rcsbuf_fill (rcsbuf, ptrend, NULL, wordp);
1670            if (ptr == NULL)
                  error (1, 0,
                         "EOF while looking for end of string in RCS file %s",
                         rcsbuf->filename);
                ptrend = rcsbuf->ptrend;
              }

              /* Handle the special case of an '' right at the end of
                 the known bytes.  */
              if (pat + 1 >= ptrend)
1680          {
                /* Note that we pass PAT, not PTR, here.  */
                pat = rcsbuf_fill (rcsbuf, pat, NULL, wordp);
                if (pat == NULL)
                {
                  /* EOF here is OK; it just means that the last
                     character of the file was an '' terminating a
                     value for a key type which does not require a
                     trailing ';'.  */
                  pat = rcsbuf->ptrend - 1;
1690
                }
                ptrend = rcsbuf->ptrend;

                /* Note that the value of PTR is bogus here.  This is
                   OK, because we don't use it.  */
              }

              if (pat + 1 >= ptrend || pat[1] != '@')
                break;
1700
              /* We found an '*/
              ++rcsbuf->embedded_at;
              ptr = pat + 2;
            }

            /* Here PAT points to the final '*/

            *pat = '\0';
```

```
        vlen = pat - *wordp;
        if (vlen == 0)
            *wordp = NULL;
        rcsbuf->vlen = vlen;
        rcsbuf->ptr = pat + 1;

        return 1;
    }

    /* C is neither ':', ';' nor '', so it should be the start of a num
       or an id.  Make sure it is not another special character. */
    if (c == '$' || c == '.' || c == ',')
    {
        error (1, 0, "illegal special character in RCS field in %s",
               rcsbuf->filename);
    }

    *wordp = ptr;
    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, wordp);
            if (ptr == NULL)
                error (1, 0, "unexpected end of file reading %s",
                       rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
        }

        /* Legitimate ID characters are digits, dots and any 'graphic
           printing character that is not a special.' This test ought
*/
        c = *ptr;
        if (isprint (c) &&
            c != ';' && c != '$' && c != ',' && c != '@' && c != ':')
        {
            ++ptr;
            continue;
        }
        break;
    }

    /* PTR points to the last non-id character in this word, and C is
       the character in its memory cell.  Check to make sure that it
       is a legitimate word delimiter - whitespace or semicolon. */
    if (c == ';' || my_whitespace (c))
    {
        rcsbuf->vlen = ptr - *wordp;
        rcsbuf->ptr = ptr;
        return 1;
    }

    error (1, 0, "illegal special character in RCS field in %s",
           rcsbuf->filename);
    /* Shut up compiler warnings.  */
    return 0;

#undef my_whitespace
}

/* Read an RCS revision number from an RCS file.  This sets *REVP to
   point to the revision number; it will point to space that is
   managed by the rcsbuf functions, and is only good until the next
   call to rcsbuf_getkey or rcsbuf_getrevnum.

   This function returns 1 on success, or 0 on EOF.  If there is an
   error reading the file, or an EOF in an unexpected location, it
   gives a fatal error.  */

static int
rcsbuf_getrevnum (rcsbuf, revp)
    struct rcsbuffer *rcsbuf;
    char **revp;
{
    char *ptr, *ptrend;
    char c;

    ptr = rcsbuf->ptr;
    ptrend = rcsbuf->ptrend;

    *revp = NULL;

    /* Skip leading whitespace.  */

    while (1)
    {
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, (char **) NULL, (char **) NULL);
```

```
            if (ptr == NULL)
1800            return 0;
            ptrend = rcsbuf->ptrend;
        }

        c = *ptr;
        if (! whitespace (c))
            break;

        ++ptr;
    }
1810
    if (! isdigit (c) && c != '.')
        error (1, 0,
               "unexpected `%c' reading revision number in RCS file %s",
               c, rcsbuf->filename);

    *revp = ptr;

    do
    {
1820    ++ptr;
        if (ptr >= ptrend)
        {
            ptr = rcsbuf_fill (rcsbuf, ptr, revp, (char **) NULL);
            if (ptr == NULL)
                error (1, 0,
                       "unexpected EOF reading revision number in RCS file %s",
                       rcsbuf->filename);
            ptrend = rcsbuf->ptrend;
        }
1830
        c = *ptr;
    }
    while (isdigit (c) || c == '.');

    if (! whitespace (c))
        error (1, 0, "unexpected `%c' reading revision number in RCS file %s",
               c, rcsbuf->filename);

    *ptr = '\0';
1840
    rcsbuf->ptr = ptr + 1;

    return 1;
}

/* Fill RCSBUF_BUFFER with bytes from the file associated with RCSBUF,
   updating PTR and the PTREND field.  If KEYP and *KEYP are not NULL,
   then *KEYP points into the buffer, and must be adjusted if the
   buffer is changed.  Likewise for VALP.  Returns the new value of
1850 PTR, or NULL on error.  */

static char *
rcsbuf_fill (rcsbuf, ptr, keyp, valp)
    struct rcsbuffer *rcsbuf;
    char *ptr;
    char **keyp;
    char **valp;
{
    int got;
1860
    if (rcsbuf->ptrend - rcsbuf_buffer + RCSBUF_BUFSIZE > rcsbuf_buffer_size)
    {
        int poff, peoff, koff, voff;

        poff = ptr - rcsbuf_buffer;
        peoff = rcsbuf->ptrend - rcsbuf_buffer;
        if (keyp != NULL && *keyp != NULL)
            koff = *keyp - rcsbuf_buffer;
        if (valp != NULL && *valp != NULL)
1870        voff = *valp - rcsbuf_buffer;

        expand_string (&rcsbuf_buffer, &rcsbuf_buffer_size,
                       rcsbuf_buffer_size + RCSBUF_BUFSIZE);

        ptr = rcsbuf_buffer + poff;
        rcsbuf->ptrend = rcsbuf_buffer + peoff;
        if (keyp != NULL && *keyp != NULL)
            *keyp = rcsbuf_buffer + koff;
        if (valp != NULL && *valp != NULL)
1880        *valp = rcsbuf_buffer + voff;
    }

    got = fread (rcsbuf->ptrend, 1, RCSBUF_BUFSIZE, rcsbuf->fp);
    if (got == 0)
    {
        if (ferror (rcsbuf->fp))
            error (1, errno, "cannot read %s", rcsbuf->filename);
        return NULL;
```

```
    }

    rcsbuf->ptrend += got;

    return ptr;
}

/* Copy the value VAL returned by rcsbuf_getkey into a memory buffer,
   returning the memory buffer.  Polish the value like
   rcsbuf_valpolish, q.v.  */

static char *
rcsbuf_valcopy (rcsbuf, val, polish, lenp)
    struct rcsbuffer *rcsbuf;
    char *val;
    int polish;
    size_t *lenp;
{
    size_t vlen;
    int embedded_at;
    char *ret;

    if (val == NULL)
    {
        if (lenp != NULL)
            *lenp = 0;
        return NULL;
    }

    vlen = rcsbuf->vlen;
    embedded_at = rcsbuf->embedded_at;

    ret = xmalloc (vlen - embedded_at + 1);

    if (rcsbuf->at_string ? embedded_at == 0 : ! polish)
    {
        /* No special action to take.  */
        memcpy (ret, val, vlen + 1);
        if (lenp != NULL)
            *lenp = vlen;
        return ret;
    }

    rcsbuf_valpolish_internal (rcsbuf, ret, val, lenp);
    return ret;
}

/* Polish the value VAL returned by rcsbuf_getkey.  The POLISH
   parameter is non-zero if multiple embedded whitespace characters
   should be compressed into a single whitespace character.  Note that
   leading and trailing whitespace was already removed by
   rcsbuf_getkey.  Within an '' string, pairs of '' characters are
   compressed into a single '' character regardless of the value of
   POLISH.  If LENP is not NULL, set *LENP to the length of the value.  */

static void
rcsbuf_valpolish (rcsbuf, val, polish, lenp)
    struct rcsbuffer *rcsbuf;
    char *val;
    int polish;
    size_t *lenp;
{
    if (val == NULL)
    {
        if (lenp != NULL)
            *lenp= 0;
        return;
    }

    if (rcsbuf->at_string ? rcsbuf->embedded_at == 0 : ! polish)
    {
        /* No special action to take.  */
        if (lenp != NULL)
            *lenp = rcsbuf->vlen;
        return;
    }

    rcsbuf_valpolish_internal (rcsbuf, val, val, lenp);
}

/* Internal polishing routine, called from rcsbuf_valcopy and
   rcsbuf_valpolish.  */

static void
rcsbuf_valpolish_internal (rcsbuf, to, from, lenp)
    struct rcsbuffer *rcsbuf;
    char *to;
    const char *from;
    size_t *lenp;
{
```

```
size_t len;

len = rcsbuf->vlen;

if (! rcsbuf->at_string)
{
   char *orig_to;
   size_t clen;

   orig_to = to;

   for (clen = len; clen > 0; ++from, --clen)
   {
      char c;

      c = *from;
      if (whitespace (c))
      {
         /* Note that we know that clen can not drop to zero
            while we have whitespace, because we know there is
            no trailing whitespace.  */
         while (whitespace (from[1]))
         {
            ++from;
            --clen;
         }
         c = ' ';
      }
      *to++ = c;
   }

   *to = '\0';

   if (lenp != NULL)
      *lenp = to - orig_to;
}
else
{
   const char *orig_from;
   char *orig_to;
   int embedded_at;
   size_t clen;

   orig_from = from;
   orig_to = to;

   embedded_at = rcsbuf->embedded_at;

   if (lenp != NULL)
      *lenp = len - embedded_at;

   for (clen = len; clen > 0; ++from, --clen)
   {
      char c;

      c = *from;
      *to++ = c;
      if (c == '@')
      {
         ++from;

         /* Sanity check.  */
         if (*from != '@' || clen == 0)
            abort ();

         --clen;

         --embedded_at;
         if (embedded_at == 0)
         {
            /* We've found all the embedded '@' characters.
               We can just memcpy the rest of the buffer after
               this '@' character.  */
            if (orig_to != orig_from)
               memcpy (to, from + 1, clen - 1);
            else
               memmove (to, from + 1, clen - 1);
            from += clen;
            to += clen - 1;
            break;
         }
      }
   }

   /* Sanity check.  */
   if (from != orig_from + len
      || to != orig_to + (len - rcsbuf->embedded_at))
   {
      abort ();
   }
```

```
2070        *to = '\0';
        }
    }

    /* Return the current position of an rcsbuf.  */

    static unsigned long
    rcsbuf_ftell (rcsbuf)
        struct rcsbuffer *rcsbuf;
    {
2080    return rcsbuf->pos + (rcsbuf->ptr - rcsbuf->buffer);
    }

    /* Return a pointer to any data buffered for RCSBUF, along with the
       length.  */

    static void
    rcsbuf_get_buffered (rcsbuf, datap, lenp)
        struct rcsbuffer *rcsbuf;
        char **datap;
2090    size_t *lenp;
    {
        *datap = rcsbuf->ptr;
        *lenp = rcsbuf->ptrend - rcsbuf->ptr;
    }

    /* CVS optimizes by quickly reading some header information from a
       file.  If it decides it needs to do more with the file, it reopens
       it.  We speed that up here by maintaining a cache of a single open
       file, to save the time it takes to reopen the file in the common
2100   case.  */

    static RCSNode *cached_rcs;
    static struct rcsbuffer cached_rcsbuf;

    /* Cache RCS and RCSBUF.  This takes responsibility for closing
       RCSBUF->FP.  */

    static void
    rcsbuf_cache (rcs, rcsbuf)
2110    RCSNode *rcs;
        struct rcsbuffer *rcsbuf;
    {
        if (cached_rcs != NULL)
            rcsbuf_cache_close ();
        cached_rcs = rcs;
        ++rcs->refcount;
        cached_rcsbuf = *rcsbuf;
    }

2120 /* If there is anything in the cache, close it.  */

    static void
    rcsbuf_cache_close ()
    {
        if (cached_rcs != NULL)
        {
            if (fclose (cached_rcsbuf.fp) != 0)
                error (0, errno, "cannot close %s", cached_rcsbuf.filename);
            rcsbuf_close (&cached_rcsbuf);
2130        freercsnode (&cached_rcs);
            cached_rcs = NULL;
        }
    }

    /* Open an rcsbuffer for RCS, getting it from the cache if possible.
       Set *FPP to the file, and *RCSBUFP to the rcsbuf.  The file should
       be put at position POS.  */

    static void
2140 rcsbuf_cache_open (rcs, pos, pfp, prcsbuf)
        RCSNode *rcs;
        long pos;
        FILE **pfp;
        struct rcsbuffer *prcsbuf;
    {
        if (cached_rcs == rcs)
        {
            if (rcsbuf_ftell (&cached_rcsbuf) != pos)
            {
2150            if (fseek (cached_rcsbuf.fp, pos, SEEK_SET) != 0)
                    error (1, 0, "cannot fseek RCS file %s",
                           cached_rcsbuf.filename);
                cached_rcsbuf.ptr = rcsbuf_buffer;
                cached_rcsbuf.ptrend = rcsbuf_buffer;
                cached_rcsbuf.pos = pos;
            }
            *pfp = cached_rcsbuf.fp;
```

```
            /* When RCS_parse opens a file using fopen_case, it frees the
               filename which we cached in CACHED_RCSBUF and stores a new
               file name in RCS->PATH.  We avoid problems here by always
               copying the filename over.  FIXME: This is hackish.  */
            cached_rcsbuf.filename = rcs->path;

            *prcsbuf = cached_rcsbuf;

            cached_rcs = NULL;

            /* Removing RCS from the cache removes a reference to it.  */
            --rcs->refcount;
            if (rcs->refcount <= 0)
                error (1, 0, "rcsbuf_cache_open: internal error");
        }
        else
        {
            if (cached_rcs != NULL)
                rcsbuf_cache_close ();

            *pfp = CVS_FOPEN (rcs->path, FOPEN_BINARY_READ);
            if (*pfp == NULL)
                error (1, 0, "unable to reopen `%s'", rcs->path);
            if (pos != 0)
            {
                if (fseek (*pfp, pos, SEEK_SET) != 0)
                    error (1, 0, "cannot fseek RCS file %s", rcs->path);
            }
            rcsbuf_open (prcsbuf, *pfp, rcs->path, pos);
        }
    }

    /*
     * process the symbols list of the rcs file
     */
    static void
    do_symbols (list, val)
        List *list;
        char *val;
    {
        Node *p;
        char *cp = val;
        char *tag, *rev;

        for (;;)
        {
            /* skip leading whitespace */
            while (whitespace (*cp))
                cp++;

            /* if we got to the end, we are done */
            if (*cp == '\0')
                break;

            /* split it up into tag and rev */
            tag = cp;
            cp = strchr (cp, ':');
            *cp++ = '\0';
            rev = cp;
            while (!whitespace (*cp) && *cp != '\0')
                cp++;
            if (*cp != '\0')
                *cp++ = '\0';

            /* make a new node and add it to the list */
            p = getnode ();
            p->key = xstrdup (tag);
            p->data = xstrdup (rev);
            (void) addnode (list, p);
        }
    }

    /*
     * process the locks list of the rcs file
     * Like do_symbols, but hash entries are keyed backwards: i.e.
     * an entry like `user:rev' is keyed on REV rather than on USER.
     */
    static void
    do_locks (list, val)
        List *list;
        char *val;
    {
        Node *p;
        char *cp = val;
        char *user, *rev;

        for (;;)
        {
            /* skip leading whitespace */
```

```
        while (whitespace (*cp))
            cp++;

        /* if we got to the end, we are done */
        if (*cp == '\0')
            break;

        /* split it up into user and rev */
        user = cp;
        cp = strchr (cp, ':');
        *cp++ = '\0';
        rev = cp;
        while (!whitespace (*cp) && *cp != '\0')
            cp++;
        if (*cp != '\0')
            *cp++ = '\0';

        /* make a new node and add it to the list */
        p = getnode ();
        p->key = xstrdup (rev);
        p->data = xstrdup (user);
        (void) addnode (list, p);
    }
}

/*
 * process the branches list of a revision delta
 */
static void
do_branches (list, val)
    List *list;
    char *val;
{
    Node *p;
    char *cp = val;
    char *branch;

    for (;;)
    {
        /* skip leading whitespace */
        while (whitespace (*cp))
            cp++;

        /* if we got to the end, we are done */
        if (*cp == '\0')
            break;

        /* find the end of this branch */
        branch = cp;
        while (!whitespace (*cp) && *cp != '\0')
            cp++;
        if (*cp != '\0')
            *cp++ = '\0';

        /* make a new node and add it to the list */
        p = getnode ();
        p->key = xstrdup (branch);
        (void) addnode (list, p);
    }
}

/*
 * process the branches list of a revision delta
 */
static void
do_remote_branches (list, val)
    List *list;
    char *val;
{
    Node *p;
    char *cp = val;
    char *branch;

    if (val == NULL)
      return;

    for (;;)
    {
        /* skip leading whitespace */
        while (whitespace (*cp))
            cp++;

        /* if we got to the end, we are done */
        if (*cp == '\0')
            break;

        /* find the end of this branch */
        branch = cp;
        while (!whitespace (*cp) && *cp != '\0')
            cp++;
```

```
        if (*cp != '\0')
            *cp++ = '\0';

        /* make a new node and add it to the list */
        p = getnode ();
        p->key = xstrdup (branch);
        (void) addnode (list, p);
    }
}

/*
 * Version Number
 *
 * Returns the requested version number of the RCS file, satisfying tags and/or
 * dates, and walking branches, if necessary.
 *
 * The result is returned; null-string if error.
 */
char *
RCS_getversion (rcs, tag, date, force_tag_match, simple_tag)
    RCSNode *rcs;
    char *tag;
    char *date;
    int force_tag_match;
    int *simple_tag;
{
    if (simple_tag != NULL)
        *simple_tag = 0;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (tag && date)
    {
        char *branch, *rev;

        if (! RCS_nodeisbranch (rcs, tag))
        {
            /* We can't get a particular date if the tag is not a
               branch.  */
            return NULL;
        }

        /* Work out the branch.  */
        if (! isdigit (tag[0]))
            branch = RCS_whatbranch (rcs, tag);
        else
            branch = xstrdup (tag);

        /* Fetch the revision of branch as of date.  */
        rev = RCS_getdatebranch (rcs, date, branch);
        free (branch);
        return (rev);
    }
    else if (tag)
        return (RCS_gettag (rcs, tag, force_tag_match, simple_tag));
    else if (date)
        return (RCS_getdate (rcs, date, force_tag_match));
    else
        return (RCS_head (rcs));

}

typedef struct {
  char*     number;
  char*     branch;
} find_remote_branch_cl;

int find_remote_branch_fn (Node* node, void* closure)
{
  find_remote_branch_cl* data = (find_remote_branch_cl*) closure;
  if (data -> branch != NULL)
    return 0;
  if (strncmp (node -> key, data -> number, strlen (data -> number)) == 0) {
    data -> branch = xstrdup (node -> key);
  }
  return 0;
}


char *
RCS_getremoteversion (finfo, rcs, tag, local_tag)
     struct file_info* finfo;
     RCSNode *rcs;
     char *tag;
     char** local_tag;
{
  *local_tag = NULL;
  RCS_fully_parse (rcs);
    assert (rcs != NULL);
```

```
    if (!isdigit (tag [0])) {
2430  /* If it's not a digit, it has to be a symbolic tag. Verify that it's a remote tag,
         to save us some errors later, and return it */
      if (strchr (tag, ':') != 0) {
        return xstrdup (tag);
      } else {
        return NULL;
      }
    } else {
      /* If it's a digit, see if the tag is on one of the remote branches. Start by finding out its
         branch magic number */
2440  if (numdots (tag) % 2 == 0) {
        /* Can't be a valid tag unless it has an odd number of dots (maybe we should allow branch tags?) */
        return NULL;
      } else {
        char* remotetag = xstrdup (tag);
        char* base = NULL;

        char* nextdot = remotetag;
        char* lastdot = NULL;
        char* beforelastdot = remotetag;
2450    int branchnum;

        for (; nextdot != NULL; nextdot = strchr (nextdot, '.')) {
          beforelastdot = lastdot + 1;
          lastdot = nextdot;
          nextdot += 1;
        }

        /* Didn't find any dots? That's no good */
        if (lastdot == NULL)
2460      return NULL;

        /* Kill the last component (we don't care) */
        *lastdot = '\0';
        /* read the branch number */
        branchnum = atoi (beforelastdot);

        /* remove the branch number */
        *beforelastdot = '\0';

2470    base = strdup (remotetag);

        /* Put in the magic branch number – note that this never
           overflows the buffer because magic branch is one char long */
        sprintf (remotetag + strlen (remotetag), "%d.%d", RCS_MAGIC_BRANCH, branchnum);

        /* Now we know the remote tag. Go down the list of remote
           branches and see if any one of them matches... */

        {
2480      Node* versionNode = NULL;
          RCSVers* version = NULL;
          find_remote_branch_cl  closure = { NULL, NULL };
          remotetag = xrealloc (remotetag, strlen (remotetag) + 2);
          strcat (remotetag, ":");
          closure.number = remotetag;
          base [strlen (base) - 1] = '\0';

          versionNode = findnode (rcs->versions, base);
          if (versionNode != NULL) {
2490        version = (RCSVers*) (versionNode -> data);
          }
          if (version != NULL) {
            walklist (version->remote_branches, find_remote_branch_fn, &closure);
          }
          free (remotetag);
          free (base);
          /* According to the RCS file, this is a remote revision
           *  However, it's possible that we already have it. Check whether we do
           */
2500      if (closure.branch == NULL) {
            return NULL;
          } else {
            char* local_rev = remote_revision_available_locally (finfo, rcs, tag);
            if (local_rev != NULL) {
              *local_tag = local_rev;
              return NULL;
            } else {
              return closure.branch;
            }
2510      }
        }
      }
    }
  }

  char* remote_revision_available_locally (struct file_info* finfo,
                                           RCSNode* rcs,
                                           char* remote_rev)
```

```
{
  char* result = 0;
  FILE* remotes_list;
  char* line;
  int line_length;
  int line_chars_allocated;

  /* Try to find it in the Remotes file */
  remotes_list = fopen (CVSADM_REMOTES, "r");
  while ((line_length = getline (&line, &line_chars_allocated, remotes_list)) > 0) {
    char* file = NULL;
    char* rev = NULL;
    char* data = NULL;

    file = line;
    rev = strchr (file, '/');
    if (rev == NULL)
      continue;

    *rev = '\0';
    rev++;

    data = strchr (rev, '/');
    if (data == NULL)
      continue;

    *data = '\0';
    data++;

    *strchr (data, '\n') = '\0';

    if ((strcmp (finfo -> file, file) == 0) && (strcmp (rev, remote_rev) == 0)) {

      /* File and revision match */
      char* fullname = xmalloc (strlen (CVSADM) + strlen (data) + 5);
      sprintf (fullname, "%s/%s", CVSADM, data);

      RCS_checkin (rcs, fullname, "Remote revision", "1.1.3", RCS_FLAGS_KEEPFILE);
      result = RCS_getbranch (rcs, xstrdup (RCS_REMOTE_BRANCH), 1);

      /* We found where the revision data is stored... */
      break;
    }
  }

  return result;
}

/*
 * Get existing revision number corresponding to tag or revision.
 * Similar to RCS_gettag but less interpretation imposed.
 * For example:
 * - If tag designates a magic branch, RCS_tag2rev
 *     returns the magic branch number.
 * - If tag is a branch tag, returns the branch number, not
 *     the revision of the head of the branch.
 * If tag or revision is not valid or does not exist in file,
 * exit with error.
 */
char *
RCS_tag2rev (rcs, tag)
    RCSNode *rcs;
    char *tag;
{
    char *rev, *pa, *pb;
    int i;

    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* If a valid revision, try to look it up */
    if ( RCS_valid_rev (tag) )
    {
        /* Make a copy so we can scribble on it */
        rev =  xstrdup (tag);

        /* If revision exists, return the copy */
        if (RCS_exist_rev (rcs, tag))
            return rev;

        /* Nope, none such. If tag is not a branch we're done. */
        i = numdots (rev);
        if ((i & 1) == 1 )
        {
            pa = strrchr (rev, '.');
            if (i == 1 || *(pa-1) != RCS_MAGIC_BRANCH || *(pa-2) != '.')
            {
                free (rev);
```

```
                error (1, 0, "revision `%s' does not exist", tag);
            }
        }

        /* Tag is branch, but does not exist, try corresponding
         * magic branch tag.
         *
         * FIXME: assumes all magic branches are of
         * form "n.n.n ... .0.n".  I'll fix if somebody can
         * send me a method to get a magic branch tag with
         * the 0 in some other position - <dangasboy.com>
*/
        pa = strrchr (rev, '.');
        pb = xmalloc (strlen (rev) + 3);
        *pa++ = 0;
        (void) sprintf (pb, "%s.%d.%s", rev, RCS_MAGIC_BRANCH, pa);
        free (rev);
        rev = pb;
        if (RCS_exist_rev (rcs, rev))
            return rev;
        error (1, 0, "revision `%s' does not exist", tag);
    }


    RCS_check_tag (tag); /* exit if not a valid tag */

    /* If tag is "HEAD", special case to get head RCS revision */
    if (tag && (strcmp (tag, TAG_HEAD) == 0))
        return (RCS_head (rcs));

    /* If valid tag let translate_symtag say yea or nay. */
    rev = translate_symtag (rcs, tag);

    if (rev)
        return rev;

    error (1, 0, "tag `%s' does not exist", tag);
    /* NOT REACHED - error (1 ... ) does not return here */
    return 0;
}

/*
 * Find the revision for a specific tag.
 * If force_tag_match is set, return NULL if an exact match is not
 * possible otherwise return RCS_head ().  We are careful to look for
 * and handle "magic" revisions specially.
 *
 * If the matched tag is a branch tag, find the head of the branch.
 *
 * Returns pointer to newly malloc'd string, or NULL.
 */
char *
RCS_gettag (rcs, symtag, force_tag_match, simple_tag)
    RCSNode *rcs;
    char *symtag;
    int force_tag_match;
    int *simple_tag;
{
    char *tag = symtag;
    int tag_allocated = 0;

    if (simple_tag != NULL)
        *simple_tag = 0;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    /* XXX this is probably not necessary, -jtc */
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* If tag is "HEAD", special case to get head RCS revision */
    if (tag && (STREQ (tag, TAG_HEAD) || *tag == '\0'))
#if 0 /* This #if 0 is only in the Cygnus code.  Why?  Death support?  */
        if (force_tag_match && (rcs->flags & VALID) && (rcs->flags & INATTIC))
            return ((char *) NULL); /* head request for removed file */
        else
#endif
            return (RCS_head (rcs));

    if (!isdigit (tag[0]))
    {
        char *version;

        /* If we got a symbolic tag, resolve it to a numeric */
        version = translate_symtag (rcs, tag);
        if (version != NULL)
        {
            int dots;
            char *magic, *branch, *cp;
```

```
2700            tag = version;
                tag_allocated = 1;

                /*
                 * If this is a magic revision, we turn it into either its
                 * physical branch equivalent (if one exists) or into
                 * its base revision, which we assume exists.
                 */
                dots = numdots (tag);
                if (dots > 2 && (dots & 1) != 0)
2710            {
                    branch = strrchr (tag, '.');
                    cp = branch++ - 1;
                    while (*cp != '.')
                        cp--;

                    /* see if we have .magic-branch. (".0.") */
                    magic = xmalloc (strlen (tag) + 1);
                    (void) sprintf (magic, ".%d.", RCS_MAGIC_BRANCH);
                    if (strncmp (magic, cp, strlen (magic)) == 0)
2720                {
                        /* it's magic.  See if the branch exists */
                        *cp = '\0';       /* turn it into a revision */
                        (void) sprintf (magic, "%s.%s", tag, branch);
                        branch = RCS_getbranch (rcs, magic, 1);
                        free (magic);
                        if (branch != NULL)
                        {
                            free (tag);
                            return (branch);
2730                    }
                        return (tag);
                    }
                    free (magic);
                }
            }
            else
            {
                /* The tag wasn't there, so return the head or NULL */
                if (force_tag_match)
2740                return (NULL);
                else
                    return (RCS_head (rcs));
            }
        }

        /*
         * numeric tag processing:
         *          1) revision number - just return it
         *          2) branch number   - find head of branch
2750     */

        /* strip trailing dots */
        while (tag[strlen (tag) - 1] == '.')
            tag[strlen (tag) - 1] = '\0';

        if ((numdots (tag) & 1) == 0)
        {
            char *branch;

2760        /* we have a branch tag, so we need to walk the branch */
            branch = RCS_getbranch (rcs, tag, force_tag_match);
            if (tag_allocated)
                free (tag);
            return branch;
        }
        else
        {
            Node *p;

2770        /* we have a revision tag, so make sure it exists */
            p = findnode (rcs->versions, tag);
            if (p != NULL)
            {
                /* We have found a numeric revision for the revision tag.
                   To support expanding the RCS keyword Name, if
                   SIMPLE_TAG is not NULL, tell the the caller that this
                   is a simple tag which co will recognize.  FIXME: Are
                   there other cases in which we should set this?  In
                   particular, what if we expand RCS keywords internally
2780               without calling co?  */
                if (simple_tag != NULL)
                    *simple_tag = 1;
                if (! tag_allocated)
                    tag = xstrdup (tag);
                return (tag);
            }
            else
            {
```

```
            /* The revision wasn't there, so return the head or NULL */
            if (tag_allocated)
                free (tag);
            if (force_tag_match)
                return (NULL);
            else
                return (RCS_head (rcs));
        }
    }
}

/*
 * Return a "magic" revision as a virtual branch off of REV for the RCS file.
 * A "magic" revision is one which is unique in the RCS file.  By unique, I
 * mean we return a revision which:
 *      - has a branch of 0 (see rcs.h RCS_MAGIC_BRANCH)
 *      - has a revision component which is not an existing branch off REV
 *      - has a revision component which is not an existing magic revision
 *      - is an even-numbered revision, to avoid conflicts with vendor branches
 * The first point is what makes it "magic".
 *
 * As an example, if we pass in 1.37 as REV, we will look for an existing
 * branch called 1.37.2.  If it did not exist, we would look for an
 * existing symbolic tag with a numeric part equal to 1.37.0.2.  If that
 * didn't exist, then we know that the 1.37.2 branch can be reserved by
 * creating a symbolic tag with 1.37.0.2 as the numeric part.
 *
 * This allows us to fork development with very little overhead – just a
 * symbolic tag is used in the RCS file.  When a commit is done, a physical
 * branch is dynamically created to hold the new revision.
 *
 * Note: We assume that REV is an RCS revision and not a branch number.
 */
static char *check_rev;
char *
RCS_magicrev (rcs, rev)
    RCSNode *rcs;
    char *rev;
{
    int rev_num;
    char *xrev, *test_branch;

    xrev = xmalloc (strlen (rev) + 14); /* enough for .0.number */
    check_rev = xrev;

    /* only look at even numbered branches */
    for (rev_num = 2; ; rev_num += 2)
    {
        /* see if the physical branch exists */
        (void) sprintf (xrev, "%s.%d", rev, rev_num);
        test_branch = RCS_getbranch (rcs, xrev, 1);
        if (test_branch != NULL) /* it did, so keep looking */
        {
            free (test_branch);
            continue;
        }

        /* now, create a "magic" revision */
        (void) sprintf (xrev, "%s.%d.%d", rev, RCS_MAGIC_BRANCH, rev_num);

        /* walk the symbols list to see if a magic one already exists */
        if (walklist (RCS_symbols(rcs), checkmagic_proc, NULL) != 0)
            continue;

        /* we found a free magic branch.  Claim it as ours */
        return (xrev);
    }
}

/*
 * walklist proc to look for a match in the symbols list.
 * Returns 0 if the symbol does not match, 1 if it does.
 */
static int
checkmagic_proc (p, closure)
    Node *p;
    void *closure;
{
    if (STREQ (check_rev, p->data))
        return (1);
    else
        return (0);
}

/*
 * Given an RCSNode, returns non-zero if the specified revision number
 * or symbolic tag resolves to a "branch" within the rcs file.
 *
 * FIXME: this is the same as RCS_nodeisbranch except for the special
 *        case for handling a null rcsnode.
```

```
 */
int
RCS_isbranch (rcs, rev)
    RCSNode *rcs;
    const char *rev;
{
    /* numeric revisions are easy - even number of dots is a branch */
    if (isdigit (*rev))
        return ((numdots (rev) & 1) == 0);

    /* assume a revision if you can't find the RCS info */
    if (rcs == NULL)
        return (0);

    /* now, look for a match in the symbols list */
    return (RCS_nodeisbranch (rcs, rev));
}

/*
 * Given an RCSNode, returns non-zero if the specified revision number
 * or symbolic tag resolves to a "branch" within the rcs file.  We do
 * take into account any magic branches as well.
 */
int
RCS_nodeisbranch (rcs, rev)
    RCSNode *rcs;
    const char *rev;
{
    int dots;
    char *version;

    assert (rcs != NULL);

    /* numeric revisions are easy - even number of dots is a branch */
    if (isdigit (*rev))
        return ((numdots (rev) & 1) == 0);

    version = translate_symtag (rcs, rev);
    if (version == NULL)
        return (0);
    dots = numdots (version);
    if ((dots & 1) == 0)
    {
        free (version);
        return (1);
    }

    /* got a symbolic tag match, but it's not a branch; see if it's magic */
    if (dots > 2)
    {
        char *magic;
        char *branch = strrchr (version, '.');
        char *cp = branch - 1;
        while (*cp != '.')
            cp--;

        /* see if we have .magic-branch. (".0.") */
        magic = xmalloc (strlen (version) + 1);
        (void) sprintf (magic, ".%d.", RCS_MAGIC_BRANCH);
        if (strncmp (magic, cp, strlen (magic)) == 0)
        {
            free (magic);
            free (version);
            return (1);
        }
        free (magic);
        free (version);
    }
    return (0);
}

/*
 * Returns a pointer to malloc'ed memory which contains the branch
 * for the specified *symbolic* tag.  Magic branches are handled correctly.
 */
char *
RCS_whatbranch (rcs, rev)
    RCSNode *rcs;
    const char *rev;
{
    char *version;
    int dots;

    /* assume no branch if you can't find the RCS info */
    if (rcs == NULL)
        return ((char *) NULL);

    /* now, look for a match in the symbols list */
    version = translate_symtag (rcs, rev);
    if (version == NULL)
```

```
        return ((char *) NULL);
    dots = numdots (version);
    if ((dots & 1) == 0)
        return (version);

    /* got a symbolic tag match, but it's not a branch; see if it's magic */
    if (dots > 2)
    {
        char *magic;
        char *branch = strrchr (version, '.');
        char *cp = branch++ - 1;
        while (*cp != '.')
            cp--;

        /* see if we have .magic-branch. (".0.") */
        magic = xmalloc (strlen (version) + 1);
        (void) sprintf (magic, ".%d.", RCS_MAGIC_BRANCH);
        if (strncmp (magic, cp, strlen (magic)) == 0)
        {
            /* yep.  it's magic.  now, construct the real branch */
            *cp = '\0';             /* turn it into a revision */
            (void) sprintf (magic, "%s.%s", version, branch);
            free (version);
            return (magic);
        }
        free (magic);
        free (version);
    }
    return ((char *) NULL);
}

/*
 * Get the head of the specified branch.  If the branch does not exist,
 * return NULL or RCS_head depending on force_tag_match.
 * Returns NULL or a newly malloc'd string.
 */
char *
RCS_getbranch (rcs, tag, force_tag_match)
    RCSNode *rcs;
    char *tag;
    int force_tag_match;
{
    Node *p, *head;
    RCSVers *vn;
    char *xtag;
    char *nextvers;
    char *cp;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* find out if the tag contains a dot, or is on the trunk */
    cp = strrchr (tag, '.');

    /* trunk processing is the special case */
    if (cp == NULL)
    {
        xtag = xmalloc (strlen (tag) + 1 + 1); /* +1 for an extra . */
        (void) strcpy (xtag, tag);
        (void) strcat (xtag, ".");
        for (cp = rcs->head; cp != NULL;)
        {
            if (strncmp (xtag, cp, strlen (xtag)) == 0)
                break;
            p = findnode (rcs->versions, cp);
            if (p == NULL)
            {
                free (xtag);
                if (force_tag_match)
                    return (NULL);
                else
                    return (RCS_head (rcs));
            }
            vn = (RCSVers *) p->data;
            cp = vn->next;
        }
        free (xtag);
        if (cp == NULL)
        {
            if (force_tag_match)
                return (NULL);
            else
                return (RCS_head (rcs));
        }
        return (xstrdup (cp));
    }
```

```
    /* if it had a '.', terminate the string so we have the base revision */
3060    *cp = '\0';

    /* look up the revision this branch is based on */
    p = findnode (rcs->versions, tag);

    /* put the . back so we have the branch again */
    *cp = '.';

    if (p == NULL)
    {
3070    /* if the base revision didn't exist, return head or NULL */
        if (force_tag_match)
            return (NULL);
        else
            return (RCS_head (rcs));
    }

    /* find the first element of the branch we are looking for */
    vn = (RCSVers *) p->data;
    if (vn->branches == NULL)
3080    return (NULL);
    xtag = xmalloc (strlen (tag) + 1 + 1); /* 1 for the extra '.' */
    (void) strcpy (xtag, tag);
    (void) strcat (xtag, ".");
    head = vn->branches->list;
    for (p = head->next; p != head; p = p->next)
        if (strncmp (p->key, xtag, strlen (xtag)) == 0)
            break;
    free (xtag);

3090    if (p == head)
    {
        /* we didn't find a match so return head or NULL */
        if (force_tag_match)
            return (NULL);
        else
            return (RCS_head (rcs));
    }

    /* now walk the next pointers of the branch */
3100    nextvers = p->key;
    do
    {
        p = findnode (rcs->versions, nextvers);
        if (p == NULL)
        {
            /* a link in the chain is missing - return head or NULL */
            if (force_tag_match)
                return (NULL);
            else
3110            return (RCS_head (rcs));
        }
        vn = (RCSVers *) p->data;
        nextvers = vn->next;
    } while (nextvers != NULL);

    /* we have the version in our hand, so go for it */
    return (xstrdup (vn->version));
}

3120 /* Returns the head of the branch which REV is on.  REV can be a
   branch tag or non-branch tag; symbolic or numeric.

   Returns a newly malloc'd string.  Returns NULL if a symbolic name
   isn't found.  */

char *
RCS_branch_head (rcs, rev)
    RCSNode *rcs;
    char *rev;
3130 {
    char *num;
    char *br;
    char *retval;

    assert (rcs != NULL);

    if (RCS_nodeisbranch (rcs, rev))
        return RCS_getbranch (rcs, rev, 1);

3140    if (isdigit (*rev))
        num = xstrdup (rev);
    else
    {
        num = translate_symtag (rcs, rev);
        if (num == NULL)
            return NULL;
    }
    br = truncate_revnum (num);
```

```
    retval = RCS_getbranch (rcs, br, 1);
    free (br);
    free (num);
    return retval;
}

/* Get the branch point for a particular branch, that is the first
   revision on that branch.  For example, RCS_getbranchpoint (rcs,
   "1.3.2") will normally return "1.3.2.1".  TARGET may be either a
   branch number or a revision number; if a revnum, find the
   branchpoint of the branch to which TARGET belongs.

   Return RCS_head if TARGET is on the trunk or if the root node could
   not be found (this is sort of backwards from our behavior on a branch;
   the rationale is that the return value is a revision from which you
   can start walking the next fields and end up at TARGET).
   Return NULL on error.  */

static char *
RCS_getbranchpoint (rcs, target)
    RCSNode *rcs;
    char *target;
{
    char *branch, *bp;
    Node *vp;
    RCSVers *rev;
    int dots, isrevnum, brlen;

    dots = numdots (target);
    isrevnum = dots & 1;

    if (dots == 1)
        /* TARGET is a trunk revision; return rcs->head. */
        return (RCS_head (rcs));

    /* Get the revision number of the node at which TARGET's branch is
       rooted.  If TARGET is a branch number, lop off the last field;
       if it's a revision number, lop off the last *two* fields. */
    branch = xstrdup (target);
    bp = strrchr (branch, '.');
    if (bp == NULL)
        error (1, 0, "%s: confused revision number %s",
               rcs->path, target);
    if (isrevnum)
        while (*--bp != '.')
            ;
    *bp = '\0';

    vp = findnode (rcs->versions, branch);
    if (vp == NULL)
    {
        error (0, 0, "%s: can't find branch point %s", rcs->path, target);
        return NULL;
    }
    rev = (RCSVers *) vp->data;

    *bp++ = '.';
    while (*bp && *bp != '.')
        ++bp;
    brlen = bp - branch;

    vp = rev->branches->list->next;
    while (vp != rev->branches->list)
    {
        /* BRANCH may be a genuine branch number, e.g. '1.1.3', or
           maybe a full revision number, e.g. '1.1.3.6'.  We have
           found our branch point if the first BRANCHLEN characters
           of the revision number match, *and* if the following
           character is a dot. */
        if (strncmp (vp->key, branch, brlen) == 0 && vp->key[brlen] == '.')
            break;
        vp = vp->next;
    }

    free (branch);
    if (vp == rev->branches->list)
    {
        error (0, 0, "%s: can't find branch point %s", rcs->path, target);
        return NULL;
    }
    else
        return (xstrdup (vp->key));
}

/*
 * Get the head of the RCS file.  If branch is set, this is the head of the
 * branch, otherwise the real head.
 * Returns NULL or a newly malloc'd string.
 */
char *
```

```
RCS_head (rcs)
    RCSNode *rcs;
{
    /* make sure we have something to look at... */
    assert (rcs != NULL);

    /*
     * NOTE: we call getbranch with force_tag_match set to avoid any
     * possibility of recursion
     */
    if (rcs->branch)
        return (RCS_getbranch (rcs, rcs->branch, 1));
    else
        return (xstrdup (rcs->head));
}

/*
 * Get the most recent revision, based on the supplied date, but use some
 * funky stuff and follow the vendor branch maybe
 */
char *
RCS_getdate (rcs, date, force_tag_match)
    RCSNode *rcs;
    char *date;
    int force_tag_match;
{
    char *cur_rev = NULL;
    char *retval = NULL;
    Node *p;
    RCSVers *vers = NULL;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* if the head is on a branch, try the branch first */
    if (rcs->branch != NULL)
        retval = RCS_getdatebranch (rcs, date, rcs->branch);

    /* if we found a match, we are done */
    if (retval != NULL)
        return (retval);

    /* otherwise if we have a trunk, try it */
    if (rcs->head)
    {
        p = findnode (rcs->versions, rcs->head);
        while (p != NULL)
        {
            /* if the date of this one is before date, take it */
            vers = (RCSVers *) p->data;
            if (RCS_datecmp (vers->date, date) <= 0)
            {
                cur_rev = vers->version;
                break;
            }

            /* if there is a next version, find the node */
            if (vers->next != NULL)
                p = findnode (rcs->versions, vers->next);
            else
                p = (Node *) NULL;
        }
    }

    /*
     * at this point, either we have the revision we want, or we have the
     * first revision on the trunk (1.1?) in our hands
     */

    /* if we found what we're looking for, and it's not 1.1 return it */
    if (cur_rev != NULL && ! STREQ (cur_rev, "1.1"))
        return (xstrdup (cur_rev));

    /* look on the vendor branch */
    retval = RCS_getdatebranch (rcs, date, CVSBRANCH);

    /*
     * if we found a match, return it; otherwise, we return the first
     * revision on the trunk or NULL depending on force_tag_match and the
     * date of the first rev
     */
    if (retval != NULL)
        return (retval);

    if (!force_tag_match || RCS_datecmp (vers->date, date) <= 0)
        return (xstrdup (vers->version));
    else
```

```
        return (NULL);
}

/*
 * Look up the last element on a branch that was put in before the specified
 * date (return the rev or NULL)
 */
static char *
RCS_getdatebranch (rcs, date, branch)
    RCSNode *rcs;
    char *date;
    char *branch;
{
    char *cur_rev = NULL;
    char *cp;
    char *xbranch, *xrev;
    Node *p;
    RCSVers *vers;

    /* look up the first revision on the branch */
    xrev = xstrdup (branch);
    cp = strrchr (xrev, '.');
    if (cp == NULL)
    {
        free (xrev);
        return (NULL);
    }
    *cp = '\0';                     /* turn it into a revision */

    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    p = findnode (rcs->versions, xrev);
    free (xrev);
    if (p == NULL)
        return (NULL);
    vers = (RCSVers *) p->data;

    /* Tentatively use this revision, if it is early enough.  */
    if (RCS_datecmp (vers->date, date) <= 0)
        cur_rev = vers->version;

    /* If no branches list, return now.  This is what happens if the branch
       is a (magic) branch with no revisions yet.  */
    if (vers->branches == NULL)
        return xstrdup (cur_rev);

    /* walk the branches list looking for the branch number */
    xbranch = xmalloc (strlen (branch) + 1 + 1); /* +1 for the extra dot */
    (void) strcpy (xbranch, branch);
    (void) strcat (xbranch, ".");
    for (p = vers->branches->list->next; p != vers->branches->list; p = p->next)
        if (strncmp (p->key, xbranch, strlen (xbranch)) == 0)
            break;
    free (xbranch);
    if (p == vers->branches->list)
    {
        /* This is what happens if the branch is a (magic) branch with
           no revisions yet.  Similar to the case where vers->branches ==
           NULL, except here there was a another branch off the same
           branchpoint.  */
        return xstrdup (cur_rev);
    }

    p = findnode (rcs->versions, p->key);

    /* walk the next pointers until you find the end, or the date is too late */
    while (p != NULL)
    {
        vers = (RCSVers *) p->data;
        if (RCS_datecmp (vers->date, date) <= 0)
            cur_rev = vers->version;
        else
            break;

        /* if there is a next version, find the node */
        if (vers->next != NULL)
            p = findnode (rcs->versions, vers->next);
        else
            p = (Node *) NULL;
    }

    /* Return whatever we found, which may be NULL.  */
    return xstrdup (cur_rev);
}

/*
 * Compare two dates in RCS format. Beware the change in format on January 1,
```

```
 * 2000, when years go from 2-digit to full format.
 */
int
RCS_datecmp (date1, date2)
    char *date1, *date2;
{
    int length_diff = strlen (date1) - strlen (date2);

    return (length_diff ? length_diff : strcmp (date1, date2));
}

/* Look up revision REV in RCS and return the date specified for the
   revision minus FUDGE seconds (FUDGE will generally be one, so that the
   logically previous revision will be found later, or zero, if we want
   the exact date).

   The return value is the date being returned as a time_t, or (time_t)-1
   on error (previously was documented as zero on error; I haven't checked
   the callers to make sure that they really check for (time_t)-1, but
   the latter is what this function really returns).  If DATE is non-NULL,
   then it must point to MAXDATELEN characters, and we store the same
   return value there in DATEFORM format.  */
time_t
RCS_getrevtime (rcs, rev, date, fudge)
    RCSNode *rcs;
    char *rev;
    char *date;
    int fudge;
{
    char tdate[MAXDATELEN];
    struct tm xtm, *ftm;
    time_t revdate = 0;
    Node *p;
    RCSVers *vers;

    /* make sure we have something to look at... */
    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* look up the revision */
    p = findnode (rcs->versions, rev);
    if (p == NULL)
        return (-1);
    vers = (RCSVers *) p->data;

    /* split up the date */
    ftm = &xtm;
    (void) sscanf (vers->date, SDATEFORM, &ftm->tm_year, &ftm->tm_mon,
                   &ftm->tm_mday, &ftm->tm_hour, &ftm->tm_min,
                   &ftm->tm_sec);

    /* If the year is from 1900 to 1999, RCS files contain only two
       digits, and sscanf gives us a year from 0-99.  If the year is
       2000+, RCS files contain all four digits and we subtract 1900,
       because the tm_year field should contain years since 1900.  */

    if (ftm->tm_year > 1900)
        ftm->tm_year -= 1900;

    /* put the date in a form getdate can grok */
    (void) sprintf (tdate, "%d/%d/%d GMT %d:%d:%d", ftm->tm_mon,
                    ftm->tm_mday, ftm->tm_year + 1900, ftm->tm_hour,
                    ftm->tm_min, ftm->tm_sec);

    /* turn it into seconds since the epoch */
    revdate = get_date (tdate, (struct timeb *) NULL);
    if (revdate != (time_t) -1)
    {
        revdate -= fudge;       /* remove "fudge" seconds */
        if (date)
        {
            /* put an appropriate string into ``date'' if we were given one */
            ftm = gmtime (&revdate);
            (void) sprintf (date, DATEFORM,
                            ftm->tm_year + (ftm->tm_year < 100 ? 0 : 1900),
                            ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
                            ftm->tm_min, ftm->tm_sec);
        }
    }
    return (revdate);
}

List *
RCS_getlocks (rcs)
    RCSNode *rcs;
{
    assert(rcs != NULL);
```

```
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (rcs->locks_data) {
        rcs->locks = getlist ();
        do_locks (rcs->locks, rcs->locks_data);
        free(rcs->locks_data);
        rcs->locks_data = NULL;
    }

    return rcs->locks;
}

List *
RCS_symbols(rcs)
    RCSNode *rcs;
{
    assert(rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (rcs->symbols_data) {
        rcs->symbols = getlist ();
        do_symbols (rcs->symbols, rcs->symbols_data);
        free(rcs->symbols_data);
        rcs->symbols_data = NULL;
    }

    return rcs->symbols;
}

/*
 * Return the version associated with a particular symbolic tag.
 * Returns NULL or a newly malloc'd string.
 */
static char *
translate_symtag (rcs, tag)
    RCSNode *rcs;
    const char *tag;
{
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (rcs->symbols != NULL)
    {
        Node *p;

        /* The symbols have already been converted into a list.  */
        p = findnode (rcs->symbols, tag);
        if (p == NULL)
            return NULL;

        return xstrdup (p->data);
    }

    if (rcs->symbols_data != NULL)
    {
        size_t len;
        char *cp;

        /* Look through the RCS symbols information.  This is like
           do_symbols, but we don't add the information to a list.  In
           most cases, we will only be called once for this file, so
           generating the list is unnecessary overhead.  */

        len = strlen (tag);
        cp = rcs->symbols_data;
        while ((cp = strchr (cp, tag[0])) != NULL)
        {
            if ((cp == rcs->symbols_data || whitespace (cp[-1]))
                && strncmp (cp, tag, len) == 0
                && cp[len] == ':')
            {
                char *v, *r;

                /* We found the tag.  Return the version number.  */

                cp += len + 1;
                v = cp;
                while (! whitespace (*cp) && *cp != '\0')
                    ++cp;
                r = xmalloc (cp - v + 1);
                strncpy (r, v, cp - v);
                r[cp - v] = '\0';
                return r;
            }

            while (! whitespace (*cp) && *cp != '\0')
                ++cp;
```

```
        }
    }

    return NULL;
}

/*
 * The argument ARG is the getopt remainder of the -k option specified on the
 * command line.  This function returns malloc'ed space that can be used
 * directly in calls to RCS V5, with the -k flag munged correctly.
 */
char *
RCS_check_kflag (arg)
    const char *arg;
{
    static const char *const  keyword_usage[] =
    {
      "%s %s: invalid RCS keyword expansion mode\n",
      "Valid expansion modes include:\n",
      "   -kkv\tGenerate keywords using the default form.\n",
      "   -kkvl\tLike -kkv, except locker's name inserted.\n",
      "   -kk\tGenerate only keyword names in keyword strings.\n",
      "   -kv\tGenerate only keyword values in keyword strings.\n",
      "   -ko\tGenerate the old keyword string (no changes from checked in file).\n",
      "   -kb\tGenerate binary file unmodified (merges not allowed) (RCS 5.7).\n",
      "(Specify the --help global option for a list of other help options)\n",
      NULL,
    };
    /* Big enough to hold any of the strings from kflags.  */
    char karg[10];
    char const *const *cpp = NULL;

    if (arg)
    {
        for (cpp = kflags; *cpp != NULL; cpp++)
        {
            if (STREQ (arg, *cpp))
                break;
        }
    }

    if (arg == NULL || *cpp == NULL)
    {
        usage (keyword_usage);
    }

    (void) sprintf (karg, "-k%s", *cpp);
    return (xstrdup (karg));
}

/*
 * Do some consistency checks on the symbolic tag... These should equate
 * pretty close to what RCS checks, though I don't know for certain.
 */
void
RCS_check_tag (tag)
    const char *tag;
{
    char *invalid = "$,.:;@";     /* invalid RCS tag characters */
    const char *cp;

    /* Check for remote tags first */
    if (tag [0] == ':') {
      const char* realtag = tag;
      realtag = strchr (realtag + 1, ':');
      if (realtag != NULL) {
        realtag = strchr (realtag + 1, ':');
        if (realtag != NULL) {
          return RCS_check_tag (realtag + 1);
        }
      }
    }

    /*
     * The first character must be an alphabetic letter. The remaining
     * characters cannot be non-visible graphic characters, and must not be
     * in the set of "invalid" RCS identifier characters.
     */
    if (isalpha (*tag))
    {
        for (cp = tag; *cp; cp++)
        {
            if (!isgraph (*cp))
                error (1, 0, "tag `%s' has non-visible graphic characters",
                       tag);
            if (strchr (invalid, *cp))
                error (1, 0, "tag `%s' must not contain the characters `%s'",
                       tag, invalid);
        }
    }
```

```
    else
        error (1, 0, "tag `%s' must start with a letter", tag);
}

/*
 * TRUE if argument has valid syntax for an RCS revision or
 * branch number.  All characters must be digits or dots, first
 * and last characters must be digits, and no two consecutive
 * characters may be dots.
 *
 * Intended for classifying things, so this function doesn't
 * call error.
 */
int
RCS_valid_rev (rev)
    char *rev;
{
    char last, c;
    last = *rev++;
    if (!isdigit (last))
        return 0;
    while ((c = *rev++))   /* Extra parens placate -Wall gcc option */
    {
        if (c == '.')
        {
            if (last == '.')
                return 0;
            continue;
        }
        last = c;
        if (!isdigit (c))
            return 0;
    }
    if (!isdigit (last))
        return 0;
    return 1;
}

/*
 * Return true if RCS revision with TAG is a dead revision.
 */
int
RCS_isdead (rcs, tag)
    RCSNode *rcs;
    const char *tag;
{
    Node *p;
    RCSVers *version;

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    p = findnode (rcs->versions, tag);
    if (p == NULL)
        return (0);

    version = (RCSVers *) p->data;
    return (version->dead);
}

/* Return the RCS keyword expansion mode.  For example "b" for binary.
   Returns a pointer into storage which is allocated and freed along with
   the rest of the RCS information; the caller should not modify this
   storage.  Returns NULL if the RCS file does not specify a keyword
   expansion mode; for all other errors, die with a fatal error.  */
char *
RCS_getexpand (rcs)
    RCSNode *rcs;
{
    assert (rcs != NULL);
    return rcs->expand;
}

/* RCS keywords, and a matching enum.  */
struct rcs_keyword
{
    const char *string;
    size_t len;
};
#define KEYWORD_INIT(s) (s), sizeof (s) - 1
static const struct rcs_keyword keywords[] =
{
    { KEYWORD_INIT ("Author") },
    { KEYWORD_INIT ("Date") },
    { KEYWORD_INIT ("Header") },
    { KEYWORD_INIT ("Id") },
    { KEYWORD_INIT ("Locker") },
    { KEYWORD_INIT ("Log") },
    { KEYWORD_INIT ("Name") },
    { KEYWORD_INIT ("RCSfile") },
```

```
    { KEYWORD_INIT ("Revision") },
    { KEYWORD_INIT ("Source") },
    { KEYWORD_INIT ("State") },
    { NULL, 0 }
};
enum keyword
{
    KEYWORD_AUTHOR = 0,
    KEYWORD_DATE,
    KEYWORD_HEADER,
    KEYWORD_ID,
    KEYWORD_LOCKER,
    KEYWORD_LOG,
    KEYWORD_NAME,
    KEYWORD_RCSFILE,
    KEYWORD_REVISION,
    KEYWORD_SOURCE,
    KEYWORD_STATE
};

/* Convert an RCS date string into a readable string.  This is like
   the RCS date2str function.  */

static char *
printable_date (rcs_date)
     const char *rcs_date;
{
    int year, mon, mday, hour, min, sec;
    char buf[100];

    (void) sscanf (rcs_date, SDATEFORM, &year, &mon, &mday, &hour, &min,
                   &sec);
    if (year < 1900)
        year += 1900;
    sprintf (buf, "%04d/%02d/%02d %02d:%02d:%02d", year, mon, mday,
             hour, min, sec);
    return xstrdup (buf);
}

/* Escape the characters in a string so that it can be included in an
   RCS value.  */

static char *
escape_keyword_value (value, free_value)
     const char *value;
     int *free_value;
{
    char *ret, *t;
    const char *s;

    for (s = value; *s != '\0'; s++)
    {
        char c;

        c = *s;
        if (c == '\t'
            || c == '\n'
            || c == '\\'
            || c == ' '
            || c == '$')
        {
            break;
        }
    }

    if (*s == '\0')
    {
        *free_value = 0;
        return (char *) value;
    }

    ret = xmalloc (strlen (value) * 4 + 1);
    *free_value = 1;

    for (s = value, t = ret; *s != '\0'; s++, t++)
    {
        switch (*s)
        {
        default:
            *t = *s;
            break;
        case '\t':
            *t++ = '\\';
            *t = 't';
            break;
        case '\n':
            *t++ = '\\';
            *t = 'n';
            break;
        case '\\':
```

```
            *t++ = '\\';
3870        *t = '\\';
            break;
        case ' ':
            *t++ = '\\';
            *t++ = '0';
            *t++ = '4';
            *t = '0';
            break;
        case '$':
            *t++ = '\\';
3880        *t++ = '0';
            *t++ = '4';
            *t = '4';
            break;
        }
    }

    *t = '\0';

    return ret;
3890 }

/* Expand RCS keywords in the memory buffer BUF of length LEN.  This
   applies to file RCS and version VERS.  If NAME is not NULL, and is
   not a numeric revision, then it is the symbolic tag used for the
   checkout.  EXPAND indicates how to expand the keywords.  This
   function sets *RETBUF and *RETLEN to the new buffer and length.
   This function may modify the buffer BUF.  If BUF != *RETBUF, then
   RETBUF is a newly allocated buffer.  */

3900 static void
expand_keywords (rcs, ver, name, log, loglen, expand, buf, len, retbuf, retlen)
     RCSNode *rcs;
     RCSVers *ver;
     const char *name;
     const char *log;
     size_t loglen;
     enum kflag expand;
     char *buf;
     size_t len;
3910  char **retbuf;
     size_t *retlen;
{
    struct expand_buffer
    {
        struct expand_buffer *next;
        char *data;
        size_t len;
        int free_data;
    } *ebufs = NULL;
3920 struct expand_buffer *ebuf_last = NULL;
    size_t ebuf_len = 0;
    char *locker;
    char *srch, *srch_next;
    size_t srch_len;

    if (expand == KFLAG_O || expand == KFLAG_B)
    {
        *retbuf = buf;
        *retlen = len;
3930    return;
    }

    /* If we are using -kkvl, dig out the locker information if any.  */
    locker = NULL;
    if (expand == KFLAG_KVL)
    {
        Node *lock;
        lock = findnode (RCS_getlocks(rcs), ver->version);
        if (lock != NULL)
3940        locker = xstrdup (lock->data);
    }

    /* RCS keywords look like $STRING$ or $STRING: VALUE$.  */
    srch = buf;
    srch_len = len;
    while ((srch_next = memchr (srch, '$', srch_len)) != NULL)
    {
        char *s, *send;
        size_t slen;
3950    const struct rcs_keyword *keyword;
        enum keyword kw;
        char *value;
        int free_value;
        char *sub;
        size_t sublen;

        srch_len -= (srch_next + 1) - srch;
        srch = srch_next + 1;
```

```
3960        /* Look for the first non alphabetic character after the '$'.  */
            send = srch + srch_len;
            for (s = srch; s < send; s++)
              if (! isalpha (*s))
                break;

            /* If the first non alphabetic character is not '$' or ':',
               then this is not an RCS keyword.  */
            if (s == send || (*s != '$' && *s != ':'))
              continue;
3970
            /* See if this is one of the keywords.  */
            slen = s - srch;
            for (keyword = keywords; keyword->string != NULL; keyword++)
            {
              if (keyword->len == slen
                  && strncmp (keyword->string, srch, slen) == 0)
              {
                break;
              }
3980        }
            if (keyword->string == NULL)
              continue;

            kw = (enum keyword) (keyword - keywords);

            /* If the keyword ends with a ':', then the old value consists
               of the characters up to the next '$'.  If there is no '$'
               before the end of the line, though, then this wasn't an RCS
               keyword after all.  */
3990        if (*s == ':')
            {
              for (; s < send; s++)
                if (*s == '$' || *s == '\n')
                  break;
              if (s == send || *s != '$')
                continue;
            }

            /* At this point we must replace the string from SRCH to S
4000           with the expansion of the keyword KW.  */

            /* Get the value to use.  */
            free_value = 0;
            if (expand == KFLAG_K)
              value = NULL;
            else
            {
              switch (kw)
              {
4010          default:
                abort ();

              case KEYWORD_AUTHOR:
                value = ver->author;
                break;

              case KEYWORD_DATE:
                value = printable_date (ver->date);
                free_value = 1;
4020            break;

              case KEYWORD_HEADER:
              case KEYWORD_ID:
                {
                  char *path;
                  int free_path;
                  char *date;

                  if (kw == KEYWORD_HEADER)
4030                path = rcs->path;
                  else
                    path = last_component (rcs->path);
                  path = escape_keyword_value (path, &free_path);
                  date = printable_date (ver->date);
                  value = xmalloc (strlen (path)
                                   + strlen (ver->version)
                                   + strlen (date)
                                   + strlen (ver->author)
                                   + strlen (ver->state)
4040                               + (locker == NULL ? 0 : strlen (locker))
                                   + 20);

                  sprintf (value, "%s %s %s %s %s%s%s",
                           path, ver->version, date, ver->author,
                           ver->state,
                           locker != NULL ? " " : "",
                           locker != NULL ? locker : "");
                  if (free_path)
```

```
                free (path);
            free (date);
            free_value = 1;
        }
        break;

    case KEYWORD_LOCKER:
        value = locker;
        break;

    case KEYWORD_LOG:
    case KEYWORD_RCSFILE:
        value = escape_keyword_value (last_component (rcs->path),
                                      &free_value);
        break;

    case KEYWORD_NAME:
        if (name != NULL && ! isdigit (*name))
            value = (char *) name;
        else
            value = NULL;
        break;

    case KEYWORD_REVISION:
        value = ver->version;
        break;

    case KEYWORD_SOURCE:
        value = escape_keyword_value (rcs->path, &free_value);
        break;

    case KEYWORD_STATE:
        value = ver->state;
        break;
    }
}

sub = xmalloc (keyword->len
               + (value == NULL ? 0 : strlen (value))
               + 10);
if (expand == KFLAG_V)
{
    /* Decrement SRCH and increment S to remove the $
       characters.  */
    --srch;
    ++srch_len;
    ++s;
    sublen = 0;
}
else
{
    strcpy (sub, keyword->string);
    sublen = strlen (keyword->string);
    if (expand != KFLAG_K)
    {
        sub[sublen] = ':';
        sub[sublen + 1] = ' ';
        sublen += 2;
    }
}
if (value != NULL)
{
    strcpy (sub + sublen, value);
    sublen += strlen (value);
}
if (expand != KFLAG_V && expand != KFLAG_K)
{
    sub[sublen] = ' ';
    ++sublen;
    sub[sublen] = '\0';
}

if (free_value)
    free (value);

/* The Log keyword requires special handling.  This behaviour
   is taken from RCS 5.7.  The special log message is what RCS
   uses for ci -k.  */
if (kw == KEYWORD_LOG
    && (sizeof "checked in with -k by " <= loglen
        || strncmp (log, "checked in with -k by ",
                    sizeof "checked in with -k by " - 1) != 0))
{
    char *start;
    char *leader;
    size_t leader_len, leader_sp_len;
    const char *logend;
    const char *snl;
    int cnl;
    char *date;
```

```
    const char *sl;

    /* We are going to insert the trailing $ ourselves, before
       the log message, so we must remove it from S, if we
       haven't done so already.  */
    if (expand != KFLAG_V)
        ++s;

    /* Find the start of the line.  */
    start = srch;
    while (start > buf && start[-1] != '\n')
        --start;

    /* Copy the start of the line to use as a comment leader.  */
    leader_len = srch - start;
    if (expand != KFLAG_V)
        --leader_len;
    leader = xmalloc (leader_len);
    memcpy (leader, start, leader_len);
    leader_sp_len = leader_len;
    while (leader_sp_len > 0 && leader[leader_sp_len - 1] == ' ')
        --leader_sp_len;

    /* RCS does some checking for an old style of Log here,
       but we don't bother.  RCS issues a warning if it
       changes anything.  */

    /* Count the number of newlines in the log message so that
       we know how many copies of the leader we will need.  */
    cnl = 0;
    logend = log + loglen;
    for (snl = log; snl < logend; snl++)
        if (*snl == '\n')
            ++cnl;

    date = printable_date (ver->date);
    sub = xrealloc (sub,
                    (sublen
                     + sizeof "Revision"
                     + strlen (ver->version)
                     + strlen (date)
                     + strlen (ver->author)
                     + loglen
                     + (cnl + 2) * leader_len
                     + 20));
    if (expand != KFLAG_V)
    {
        sub[sublen] = '$';
        ++sublen;
    }
    sub[sublen] = '\n';
    ++sublen;
    memcpy (sub + sublen, leader, leader_len);
    sublen += leader_len;
    sprintf (sub + sublen, "Revision %s  %s  %s\n",
             ver->version, date, ver->author);
    sublen += strlen (sub + sublen);
    free (date);

    sl = log;
    while (sl < logend)
    {
        if (*sl == '\n')
        {
            memcpy (sub + sublen, leader, leader_sp_len);
            sublen += leader_sp_len;
            sub[sublen] = '\n';
            ++sublen;
            ++sl;
        }
        else
        {
            const char *slnl;

            memcpy (sub + sublen, leader, leader_len);
            sublen += leader_len;
            for (slnl = sl; slnl < logend && *slnl != '\n'; ++slnl)
                ;
            if (slnl < logend)
                ++slnl;
            memcpy (sub + sublen, sl, slnl - sl);
            sublen += slnl - sl;
            sl = slnl;
        }
    }

    memcpy (sub + sublen, leader, leader_sp_len);
    sublen += leader_sp_len;

    free (leader);
```

```
        }
4230
        /* Now SUB contains a string which is to replace the string
           from SRCH to S.  SUBLEN is the length of SUB.  */

        if (srch + sublen == s)
        {
            memcpy (srch, sub, sublen);
            free (sub);
        }
        else
4240    {
            struct expand_buffer *ebuf;

            /* We need to change the size of the buffer.  We build a
               list of expand_buffer structures.  Each expand_buffer
               structure represents a portion of the final output.  We
               concatenate them back into a single buffer when we are
               done.  This minimizes the number of potentially large
               buffer copies we must do.  */

4250        if (ebufs == NULL)
            {
                ebufs = (struct expand_buffer *) xmalloc (sizeof *ebuf);
                ebufs->next = NULL;
                ebufs->data = buf;
                ebufs->free_data = 0;
                ebuf_len = srch - buf;
                ebufs->len = ebuf_len;
                ebuf_last = ebufs;
            }
4260        else
            {
                assert (srch >= ebuf_last->data);
                assert (srch <= ebuf_last->data + ebuf_last->len);
                ebuf_len -= ebuf_last->len - (srch - ebuf_last->data);
                ebuf_last->len = srch - ebuf_last->data;
            }

            ebuf = (struct expand_buffer *) xmalloc (sizeof *ebuf);
            ebuf->data = sub;
4270        ebuf->len = sublen;
            ebuf->free_data = 1;
            ebuf->next = NULL;
            ebuf_last->next = ebuf;
            ebuf_last = ebuf;
            ebuf_len += sublen;

            ebuf = (struct expand_buffer *) xmalloc (sizeof *ebuf);
            ebuf->data = s;
            ebuf->len = srch_len - (s - srch);
4280        ebuf->free_data = 0;
            ebuf->next = NULL;
            ebuf_last->next = ebuf;
            ebuf_last = ebuf;
            ebuf_len += srch_len - (s - srch);
        }

        srch_len -= (s - srch);
        srch = s;
    }
4290
    if (locker != NULL)
        free (locker);

    if (ebufs == NULL)
    {
        *retbuf = buf;
        *retlen = len;
    }
    else
4300{
        char *ret;

        ret = xmalloc (ebuf_len);
        *retbuf = ret;
        *retlen = ebuf_len;
        while (ebufs != NULL)
        {
            struct expand_buffer *next;

4310        memcpy (ret, ebufs->data, ebufs->len);
            ret += ebufs->len;
            if (ebufs->free_data)
                free (ebufs->data);
            next = ebufs->next;
            free (ebufs);
            ebufs = next;
        }
    }
```

```
}

/* Check out a revision from an RCS file.

   If PFN is not NULL, then ignore WORKFILE and SOUT.  Call PFN zero
   or more times with the contents of the file.  CALLERDAT is passed,
   uninterpreted, to PFN.  (The current code will always call PFN
   exactly once for a non empty file; however, the current code
   assumes that it can hold the entire file contents in memory, which
   is not a good assumption, and might change in the future).

   Otherwise, if WORKFILE is not NULL, check out the revision to
   WORKFILE.  However, if WORKFILE is not NULL, and noexec is set,
   then don't do anything.

   Otherwise, if WORKFILE is NULL, check out the revision to SOUT.  If
   SOUT is RUN_TTY, then write the contents of the revision to
   standard output.  When using SOUT, the output is generally a
   temporary file; don't bother to get the file modes correct.

   REV is the numeric revision to check out.  It may be NULL, which
   means to check out the head of the default branch.

   If NAMETAG is not NULL, and is not a numeric revision, then it is
   the tag that should be used when expanding the RCS Name keyword.

   OPTIONS is a string such as "-kb" or "-kv" for keyword expansion
   options.  It may be NULL to use the default expansion mode of the
   file, typically "-kkv".

   On an error which prevented checking out the file, either print a
   nonfatal error and return 1, or give a fatal error.  On success,
   return 0.  */

/* This function mimics the behavior of 'rcs co' almost exactly.  The
   chief difference is in its support for preserving file ownership,
   permissions, and special files across checkin and checkout - see
   comments in RCS_checkin for some issues about this. -twp */

int
RCS_checkout (rcs, workfile, rev, nametag, options, sout, pfn, callerdat)
     RCSNode *rcs;
     char *workfile;
     char *rev;
     char *nametag;
     char *options;
     char *sout;
     RCSCHECKOUTPROC pfn;
     void *callerdat;
{
    int free_rev = 0;
    enum kflag expand;
    FILE *fp, *ofp;
    struct stat sb;
    struct rcsbuffer rcsbuf;
    char *key;
    char *value;
    size_t len;
    int free_value = 0;
    char *log = NULL;
    size_t loglen;
    Node *vp = NULL;
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    uid_t rcs_owner = (uid_t) -1;
    gid_t rcs_group = (gid_t) -1;
    mode_t rcs_mode;
    int change_rcs_owner_or_group = 0;
    int change_rcs_mode = 0;
    int special_file = 0;
    unsigned long devnum_long;
    dev_t devnum = 0;
#endif

    if (trace)
    {
        (void) fprintf (stderr, "%s-> checkout (%s, %s, %s, %s)\n",
#ifdef SERVER_SUPPORT
                        server_active ? "S" : " ",
#else
                        "",
#endif
                        rcs->path,
                        rev != NULL ? rev : "",
                        options != NULL ? options : "",
                        (pfn != NULL ? "(function)"
                         : (workfile != NULL
                            ? workfile
                            : (sout != RUN_TTY ? sout : "(stdout)"))));
    }
```

```
    assert (rev == NULL || isdigit (*rev));

    if (noexec && workfile != NULL)
        return 0;

    assert (sout == RUN_TTY || workfile == NULL);
    assert (pfn == NULL || (sout == RUN_TTY && workfile == NULL));

    /* Some callers, such as Checkin or remove_file, will pass us a
       branch.  */
    if (rev != NULL && (numdots (rev) & 1) == 0)
    {
        rev = RCS_getbranch (rcs, rev, 1);
        if (rev == NULL)
            error (1, 0, "internal error: bad branch tag in checkout");
        free_rev = 1;
    }

    if (rev == NULL || STREQ (rev, rcs->head))
    {
        int gothead;

        /* We want the head revision.  Try to read it directly.  */

        if (rcs->flags & PARTIAL)
            RCS_reparsercsfile (rcs, &fp, &rcsbuf);
        else
            rcsbuf_cache_open (rcs, rcs->delta_pos, &fp, &rcsbuf);

        gothead = 0;
        if (! rcsbuf_getrevnum (&rcsbuf, &key))
            error (1, 0, "unexpected EOF reading %s", rcs->path);
        while (rcsbuf_getkey (&rcsbuf, &key, &value))
        {
            if (STREQ (key, "log"))
                log = rcsbuf_valcopy (&rcsbuf, value, 0, &loglen);
            else if (STREQ (key, "text"))
            {
                gothead = 1;
                break;
            }
        }

        if (! gothead)
        {
            error (0, 0, "internal error: cannot find head text");
            if (free_rev)
                free (rev);
            return 1;
        }

        rcsbuf_valpolish (&rcsbuf, value, 0, &len);

        if (fstat (fileno (fp), &sb) < 0)
            error (1, errno, "cannot fstat %s", rcs->path);

        rcsbuf_cache (rcs, &rcsbuf);
    }
    else
    {
        struct rcsbuffer *rcsbufp;

        /* It isn't the head revision of the trunk.  We'll need to
           walk through the deltas.  */

        fp = NULL;
        if (rcs->flags & PARTIAL)
            RCS_reparsercsfile (rcs, &fp, &rcsbuf);

        if (fp == NULL)
        {
            /* If RCS_deltas didn't close the file, we could use fstat
               here too.  Probably should change it thusly....  */
            if (stat (rcs->path, &sb) < 0)
                error (1, errno, "cannot stat %s", rcs->path);
            rcsbufp = NULL;
        }
        else
        {
            if (fstat (fileno (fp), &sb) < 0)
                error (1, errno, "cannot fstat %s", rcs->path);
            rcsbufp = &rcsbuf;
        }

        RCS_deltas (rcs, fp, rcsbufp, rev, RCS_FETCH, &value, &len,
                    &log, &loglen);
        free_value = 1;
    }

    /* If OPTIONS is NULL or the empty string, then the old code would
```

```
        invoke the RCS co program with no -k option, which means that
4500    co would use the string we have stored in rcs->expand.  */
    if ((options == NULL || options[0] == '\0') && rcs->expand == NULL)
        expand = KFLAG_KV;
    else
    {
        const char *ouroptions;
        const char * const *cpp;

        if (options != NULL && options[0] != '\0')
        {
4510        assert (options[0] == '-' && options[1] == 'k');
            ouroptions = options + 2;
        }
        else
            ouroptions = rcs->expand;

        for (cpp = kflags; *cpp != NULL; cpp++)
            if (STREQ (*cpp, ouroptions))
                break;

4520    if (*cpp != NULL)
            expand = (enum kflag) (cpp - kflags);
        else
        {
            error (0, 0,
                   "internal error: unsupported substitution string -k%s",
                   ouroptions);
            expand = KFLAG_KV;
        }
    }
4530
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* Handle special files and permissions, if that is desired. */
    if (preserve_perms)
    {
        RCSVers *vers;
        Node *info;

        vp = findnode (rcs->versions, rev == NULL ? rcs->head : rev);
        if (vp == NULL)
4540        error (1, 0, "internal error: no revision information for %s",
                   rev == NULL ? rcs->head : rev);
        vers = (RCSVers *) vp->data;

        /* First we look for symlinks, which are simplest to handle. */
        info = findnode (vers->other_delta, "symlink");
        if (info != NULL)
        {
            char *dest;

4550        if (pfn != NULL || (workfile == NULL && sout == RUN_TTY))
                error (1, 0, "symbolic link %s:%s cannot be piped",
                       rcs->path, vers->version);
            if (workfile == NULL)
                dest = sout;
            else
                dest = workfile;

            /* Remove 'dest', just in case.  It's okay to get ENOENT here,
               since we just want the file not to be there.  (TODO: decide
4560           whether it should be considered an error for 'dest' to exist
               at this point.  If so, the unlink call should be removed and
               'symlink' should signal the error. -twp) */
            if (unlink (dest) < 0 && !existence_error (errno))
                error (1, errno, "cannot remove %s", dest);
            if (symlink (info->data, dest) < 0)
                error (1, errno, "cannot create symbolic link from %s to %s",
                       dest, info->data);
            if (free_value)
                free (value);
4570        if (free_rev)
                free (rev);
            return 0;
        }

        /* Next, we look at this file's hardlinks field, and see whether
           it is linked to any other file that has been checked out.
           If so, we don't do anything else - just link it to that file.

           If we are checking out a file to a pipe or temporary storage,
4580       none of this should matter.  Hence the 'workfile != NULL'
           wrapper around the whole thing. -twp */

        if (workfile != NULL)
        {
            List *links = vers->hardlinks;
            if (links != NULL)
            {
                Node *uptodate_link;
```

```
4590                /* For each file in the hardlinks field, check to see
                       if it exists, and if so, if it has been checked out
                       this iteration.  When walklist returns, uptodate_link
                       should point to a hardlist node representing a file
                       in 'links' which has recently been checked out, or
                       NULL if no file in 'links' has yet been checked out. */

                    uptodate_link = NULL;
                    (void) walklist (links, find_checkedout_proc, &uptodate_link);
                    dellist (&links);
4600
                    /* If we've found a file that 'workfile' is supposed to be
                       linked to, and it has been checked out since CVS was
                       invoked, then simply link workfile to that file and return.

                       If one of these conditions is not met, then
                       workfile is the first one in its hardlink group to
                       be checked out, and we must continue with a full
                       checkout. */

4610                if (uptodate_link != NULL)
                    {
                        struct hardlink_info *hlinfo =
                            (struct hardlink_info *) uptodate_link->data;

                        if (link (uptodate_link->key, workfile) < 0)
                            error (1, errno, "cannot link %s to %s",
                                   workfile, uptodate_link->key);
                        hlinfo->checked_out = 1; /* probably unnecessary */
                        if (free_value)
4620                        free (value);
                        if (free_rev)
                            free (rev);
                        return 0;
                    }
                }
            }

            info = findnode (vers->other_delta, "owner");
            if (info != NULL)
4630        {
                change_rcs_owner_or_group = 1;
                rcs_owner = (uid_t) strtoul (info->data, NULL, 10);
            }
            info = findnode (vers->other_delta, "group");
            if (info != NULL)
            {
                change_rcs_owner_or_group = 1;
                rcs_group = (gid_t) strtoul (info->data, NULL, 10);
            }
4640        info = findnode (vers->other_delta, "permissions");
            if (info != NULL)
            {
                change_rcs_mode = 1;
                rcs_mode = (mode_t) strtoul (info->data, NULL, 8);
            }
            info = findnode (vers->other_delta, "special");
            if (info != NULL)
            {
                /* If the size of 'devtype' changes, fix the sscanf call also */
4650            char devtype[16];

                if (sscanf (info->data, "%16s %lu",
                            devtype, &devnum_long) < 2)
                    error (1, 0, "%s:%s has bad 'special' newphrase %s",
                           workfile, vers->version, info->data);
                devnum = devnum_long;
                if (strcmp (devtype, "character") == 0)
                    special_file = S_IFCHR;
                else if (strcmp (devtype, "block") == 0)
4660                special_file = S_IFBLK;
                else
                    error (0, 0, "%s is a special file of unsupported type '%s'",
                           workfile, info->data);
            }
        }
    #endif

        if (expand != KFLAG_O && expand != KFLAG_B)
        {
4670        char *newvalue;

            /* Don't fetch the delta node again if we already have it. */
            if (vp == NULL)
            {
                vp = findnode (rcs->versions, rev == NULL ? rcs->head : rev);
                if (vp == NULL)
                    error (1, 0, "internal error: no revision information for %s",
                           rev == NULL ? rcs->head : rev);
```

```
        }

        expand_keywords (rcs, (RCSVers *) vp->data, nametag, log, loglen,
                         expand, value, len, &newvalue, &len);

        if (newvalue != value)
        {
            if (free_value)
                free (value);
            value = newvalue;
            free_value = 1;
        }
    }

    if (free_rev)
        free (rev);

    if (log != NULL)
    {
        free (log);
        log = NULL;
    }

    if (pfn != NULL)
    {
#ifdef PRESERVE_PERMISSIONS_SUPPORT
        if (special_file)
            error (1, 0, "special file %s cannot be piped to anything",
                   rcs->path);
#endif
        /* The PFN interface is very simple to implement right now, as
           we always have the entire file in memory.  */
        if (len != 0)
            pfn (callerdat, value, len);
    }
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    else if (special_file)
    {
        char *dest;

        /* Can send either to WORKFILE or to SOUT, as long as SOUT is
           not RUN_TTY. */
        dest = workfile;
        if (dest == NULL)
        {
            if (sout == RUN_TTY)
                error (1, 0, "special file %s cannot be written to stdout",
                       rcs->path);
            dest = sout;
        }

        /* Unlink 'dest', just in case.  It's okay if this provokes a
           ENOENT error. */
        if (unlink (dest) < 0 && existence_error (errno))
            error (1, errno, "cannot remove %s", dest);
        if (mknod (dest, special_file, devnum) < 0)
            error (1, errno, "could not create special file %s",
                   dest);
    }
#endif
    else
    {
        /* Not a special file: write to WORKFILE or SOUT. */
        if (workfile == NULL)
        {
            if (sout == RUN_TTY)
                ofp = stdout;
            else
            {
                /* Symbolic links should be removed before replacement, so that
                   'fopen' doesn't follow the link and open the wrong file. */
                if (islink (sout))
                    if (unlink_file (sout) < 0)
                        error (1, errno, "cannot remove %s", sout);
                ofp = CVS_FOPEN (sout, expand == KFLAG_B ? "wb" : "w");
                if (ofp == NULL)
                    error (1, errno, "cannot open %s", sout);
            }
        }
        else
        {
            /* Output is supposed to go to WORKFILE, so we should open that
               file.  Symbolic links should be removed first (see above). */
            if (islink (workfile))
                if (unlink_file (workfile) < 0)
                    error (1, errno, "cannot remove %s", workfile);

            ofp = CVS_FOPEN (workfile, expand == KFLAG_B ? "wb" : "w");

            /* If the open failed because the existing workfile was not
```

```
                writable, try to chmod the file and retry the open.  */
4770            if (ofp == NULL && errno == EACCES
                    && isfile (workfile) && !iswritable (workfile))
                {
                    xchmod (workfile, 1);
                    ofp = CVS_FOPEN (workfile, expand == KFLAG_B ? "wb" : "w");
                }

                if (ofp == NULL)
                {
                    error (0, errno, "cannot open %s", workfile);
4780                if (free_value)
                        free (value);
                    return 1;
                }
            }

            if (workfile == NULL && sout == RUN_TTY)
            {
                if (expand == KFLAG_B)
                    cvs_output_binary (value, len);
4790            else
                {
                    /* cvs_output requires the caller to check for zero
                       length.  */
                    if (len > 0)
                        cvs_output (value, len);
                }
            }
            else
            {
4800            /* NT 4.0 is said to have trouble writing 2099999 bytes
                   (for example) in a single fwrite.  So break it down
                   (there is no need to be writing that much at once
                   anyway; it is possible that LARGEST_FWRITE should be
                   somewhat larger for good performance, but for testing I
                   want to start with a small value until/unless a bigger
                   one proves useful).  */
#define LARGEST_FWRITE 8192
                size_t nleft = len;
                size_t nstep = (len < LARGEST_FWRITE ? len : LARGEST_FWRITE);
4810            char *p = value;

                while (nleft > 0)
                {
                    if (fwrite (p, 1, nstep, ofp) != nstep)
                    {
                        error (0, errno, "cannot write %s",
                               (workfile != NULL
                                ? workfile
                                : (sout != RUN_TTY ? sout : "stdout")));
4820                    if (free_value)
                            free (value);
                        return 1;
                    }
                    p += nstep;
                    nleft -= nstep;
                    if (nleft < nstep)
                        nstep = nleft;
                }
            }
4830    }

        if (free_value)
            free (value);

        if (workfile != NULL)
        {
            int ret;

#ifdef PRESERVE_PERMISSIONS_SUPPORT
4840        if (!special_file && fclose (ofp) < 0)
            {
                error (0, errno, "cannot close %s", workfile);
                return 1;
            }

            if (change_rcs_owner_or_group)
            {
                if (chown (workfile, rcs_owner, rcs_group) < 0)
                    error (0, errno, "could not change owner or group of %s",
4850                       workfile);
            }

            ret = chmod (workfile,
                         change_rcs_mode
                         ? rcs_mode
                         : sb.st_mode & ~(S_IWRITE | S_IWGRP | S_IWOTH));
#else
            if (fclose (ofp) < 0)
```

```
        {
4860        error (0, errno, "cannot close %s", workfile);
            return 1;
        }

        ret = chmod (workfile,
                     sb.st_mode & ~(S_IWRITE | S_IWGRP | S_IWOTH));
#endif
        if (ret < 0)
        {
            error (0, errno, "cannot change mode of file %s",
4870               workfile);
        }
    }
    else if (sout != RUN_TTY)
    {
        if (
#ifdef PRESERVE_PERMISSIONS_SUPPORT
            !special_file &&
#endif
            fclose (ofp) < 0)
4880    {
            error (0, errno, "cannot close %s", sout);
            return 1;
        }
    }

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* If we are in the business of preserving hardlinks, then
       mark this file as having been checked out. */
    if (preserve_perms && workfile != NULL)
4890    update_hardlink_info (workfile);
#endif

    return 0;
}

static RCSVers *RCS_findlock_or_tip PROTO ((RCSNode *rcs));

/* Find the delta currently locked by the user.  From the 'ci' man page:

4900    "If rev is omitted, ci tries to derive the new revision
        number from the caller's last lock. If the caller has
        locked the tip revision of a branch, the new revision is
        appended to that branch. The new revision number is
        obtained by incrementing the tip revision number. If the
        caller locked a non-tip revision, a new branch is started
        at that revision by incrementing the highest branch number
        at that revision. The default initial branch and level
        numbers are 1.

4910    If rev is omitted and the caller has no lock, but owns the
        file and locking is not set to strict, then the revision
        is appended to the default branch (normally the trunk; see
        the -b option of rcs(1))."

   RCS_findlock_or_tip finds the unique revision locked by the caller
   and returns its delta node.  If the caller has not locked any
   revisions (and is permitted to commit to an unlocked delta, as
   described above), return the tip of the default branch. */

4920 static RCSVers *
RCS_findlock_or_tip (rcs)
    RCSNode *rcs;
{
    char *user = getcaller();
    Node *lock, *p;
    List *locklist;

    /* Find unique delta locked by caller. This code is very similar
       to the code in RCS_unlock - perhaps it could be abstracted
4930   into a RCS_findlock function. */
    locklist = RCS_getlocks (rcs);
    lock = NULL;
    for (p = locklist->list->next; p != locklist->list; p = p->next)
    {
        if (STREQ (p->data, user))
        {
            if (lock != NULL)
            {
                error (0, 0, "\
4940 %s: multiple revisions locked by %s; please specify one", rcs->path, user);
                return NULL;
            }
            lock = p;
        }
    }

    if (lock != NULL)
    {
```

```
        /* Found an old lock, but check that the revision still exists. */
        p = findnode (rcs->versions, lock->key);
        if (p == NULL)
        {
            error (0, 0, "%s: can't unlock nonexistent revision %s",
                   rcs->path,
                   lock->key);
            return NULL;
        }
        return (RCSVers *) p->data;
    }

    /* No existing lock.  The RCS rule is that this is an error unless
       locking is nonstrict AND the file is owned by the current
       user.  Trying to determine the latter is a portability nightmare
       in the face of NT, VMS, AFS, and other systems with non-unix-like
       ideas of users and owners.  In the case of CVS, we should never get
       here (as long as the traditional behavior of making sure to call
       RCS_lock persists).  Anyway, we skip the RCS error checks
       and just return the default branch or head.  The reasoning is that
       those error checks are to make users lock before a checkin, and we do
       that in other ways if at all anyway (e.g. rcslock.pl).  */

    p = findnode (rcs->versions, RCS_getbranch (rcs, rcs->branch, 0));
    return (RCSVers *) p->data;
}

/* Revision number string, R, must contain a '.'.
   Return a newly-malloc'd copy of the prefix of R up
   to but not including the final '.'.  */

static char *
truncate_revnum (r)
    const char *r;
{
    size_t len;
    char *new_r;
    char *dot = strrchr (r, '.');

    assert (dot);
    len = dot - r;
    new_r = xmalloc (len + 1);
    memcpy (new_r, r, len);
    *(new_r + len) = '\0';
    return new_r;
}

/* Revision number string, R, must contain a '.'.
   R must be writable.  Replace the rightmost '.' in R with
   the NUL byte and return a pointer to that NUL byte.  */

static char *
truncate_revnum_in_place (r)
    char *r;
{
    char *dot = strrchr (r, '.');
    assert (dot);
    *dot = '\0';
    return dot;
}

/* Revision number strings, R and S, must each contain a '.'.
   R and S must be writable and must have the same number of dots.
   Truncate R and S for the comparison, then restored them to their
   original state.
   Return the result (see compare_revnums) of comparing R and S
   ignoring differences in any component after the rightmost '.'.  */

static int
compare_truncated_revnums (r, s)
    char *r;
    char *s;
{
    char *r_dot = truncate_revnum_in_place (r);
    char *s_dot = truncate_revnum_in_place (s);
    int cmp;

    assert (numdots (r) == numdots (s));

    cmp = compare_revnums (r, s);

    *r_dot = '.';
    *s_dot = '.';

    return cmp;
}

/* Return a malloc'd copy of the string representing the highest branch
   number on BRANCHNODE.  If there are no branches on BRANCHNODE, return NULL.
   FIXME: isn't the max rev always the last one?
```

```
   If so, we don't even need a loop.  */
5040
static char *max_rev PROTO ((const RCSVers *));

static char *
max_rev (branchnode)
    const RCSVers *branchnode;
{
    Node *head;
    Node *bp;
    char *max;
5050

    if (branchnode->branches == NULL)
    {
        return NULL;
    }

    max = NULL;
    head = branchnode->branches->list;
    for (bp = head->next; bp != head; bp = bp->next)
    {
5060    if (max == NULL || compare_truncated_revnums (max, bp->key) < 0)
        {
            max = bp->key;
        }
    }
    assert (max);

    return truncate_revnum (max);
}

5070 /* Create BRANCH in RCS's delta tree.  BRANCH may be either a branch
   number or a revision number.  In the former case, create the branch
   with the specified number; in the latter case, create a new branch
   rooted at node BRANCH with a higher branch number than any others.
   Return the number of the tip node on the new branch. */

static char *
RCS_addbranch (rcs, branch)
    RCSNode *rcs;
    const char *branch;
5080 {
    char *branchpoint, *newrevnum;
    Node *nodep, *bp;
    Node *marker;
    RCSVers *branchnode;

    /* Append to end by default.  */
    marker = NULL;

    branchpoint = xstrdup (branch);
5090 if ((numdots (branchpoint) & 1) == 0)
    {
        truncate_revnum_in_place (branchpoint);
    }

    /* Find the branch rooted at BRANCHPOINT. */
    nodep = findnode (rcs->versions, branchpoint);
    if (nodep == NULL)
    {
        error (0, 0, "%s: can't find branch point %s", rcs->path, branchpoint);
5100    return NULL;
    }
    branchnode = (RCSVers *) nodep->data;

    /* If BRANCH was a full branch number, make sure it is higher than MAX. */
    if ((numdots (branch) & 1) == 1)
    {
        if (branchnode->branches == NULL)
        {
            /* We have to create the first branch on this node, which means
5110           appending ".2" to the revision number. */
            newrevnum = (char *) xmalloc (strlen (branch) + 3);
            strcpy (newrevnum, branch);
            strcat (newrevnum, ".2");
        }
        else
        {
            char *max = max_rev (branchnode);
            assert (max);
            newrevnum = increment_revnum (max);
5120        free (max);
        }
    }
    else
    {
        newrevnum = xstrdup (branch);

        if (branchnode->branches != NULL)
        {
```

```
            Node *head;
5130        Node *bp;

            /* Find the position of this new branch in the sorted list
               of branches. */
            head = branchnode->branches->list;
            for (bp = head->next; bp != head; bp = bp->next)
            {
                char *dot;
                int found_pos;

5140            /* The existing list must be sorted on increasing revnum. */
                assert (bp->next == head
                        || compare_truncated_revnums (bp->key,
                                                      bp->next->key) < 0);
                dot = truncate_revnum_in_place (bp->key);
                found_pos = (compare_revnums (branch, bp->key) < 0);
                *dot = '.';

                if (found_pos)
                {
5150                break;
                }
            }
            marker = bp;
        }
    }

    newrevnum = (char *) xrealloc (newrevnum, strlen (newrevnum) + 3);
    strcat (newrevnum, ".1");

5160    /* Add this new revision number to BRANCHPOINT's branches list. */
    if (branchnode->branches == NULL)
        branchnode->branches = getlist();
    bp = getnode();
    bp->key = xstrdup (newrevnum);

    /* Append to the end of the list by default, that is, just before
       the header node, 'list'. */
    if (marker == NULL)
        marker = branchnode->branches->list;
5170
    {
        int fail;
        fail = insert_before (branchnode->branches, marker, bp);
        assert (!fail);
    }

    return newrevnum;
}

5180 /* Check in to RCSFILE with revision REV (which must be greater than
    the largest revision) and message MESSAGE (which is checked for
    legality). If FLAGS & RCS_FLAGS_DEAD, check in a dead revision.
    If FLAGS & RCS_FLAGS_QUIET, tell ci to be quiet. If FLAGS &
    RCS_FLAGS_MODTIME, use the working file's modification time for the
    checkin time. WORKFILE is the working file to check in from, or
    NULL to use the usual RCS rules for deriving it from the RCSFILE.
    If FLAGS & RCS_FLAGS_KEEPFILE, don't unlink the working file;
    unlinking the working file is standard RCS behavior, but is rarely
    appropriate for CVS.
5190
    This function should almost exactly mimic the behavior of 'rcs ci'. The
    principal point of difference is the support here for preserving file
    ownership and permissions in the delta nodes. This is not a clean
    solution - precisely because it diverges from RCS's behavior - but
    it doesn't seem feasible to do this anywhere else in the code. [-twp]

    Return value is -1 for error (and errno is set to indicate the
    error), positive for error (and an error message has been printed),
    or zero for success. */
5200
int
RCS_checkin (rcs, workfile, message, rev, flags)
    RCSNode *rcs;
    char *workfile;
    char *message;
    char *rev;
    int flags;
{
    RCSVers *delta, *commitpt;
5210    Deltatext *dtext;
    Node *nodep;
    char *tmpfile, *changefile, *chtext;
    char *diffopts;
    size_t bufsize;
    int buflen, chtextlen;
    int status, checkin_quiet, allocated_workfile;
    struct tm *ftm;
    time_t modtime;
```

```
    int adding_branch = 0;

    commitpt = NULL;

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* Get basename of working file.  Is there a library function to
       do this?  I couldn't find one. -twp */
    allocated_workfile = 0;
    if (workfile == NULL)
    {
        char *p;
        int extlen = strlen (RCSEXT);
        workfile = xstrdup (last_component (rcs->path));
        p = workfile + (strlen (workfile) - extlen);
        assert (strncmp (p, RCSEXT, extlen) == 0);
        *p = '\0';
        allocated_workfile = 1;
    }

    /* Is the backend file a symbolic link?  Follow it and replace the
       filename with the destination of the link.  */

    while (islink (rcs->path))
    {
        char *newname;
#ifdef HAVE_READLINK
        /* The clean thing to do is probably to have each filesubr.c
           implement this (with an error if not supported by the
           platform, in which case islink would presumably return 0).
           But that would require editing each filesubr.c and so the
           expedient hack seems to be looking at HAVE_READLINK.  */
        newname = xreadlink (rcs->path);
#else
        error (1, 0, "internal error: islink doesn't like readlink");
#endif

        if (isabsolute (newname))
        {
            free (rcs->path);
            rcs->path = newname;
        }
        else
        {
            char *oldname = last_component (rcs->path);
            int dirlen = oldname - rcs->path;
            char *fullnewname = xmalloc (dirlen + strlen (newname) + 1);
            strncpy (fullnewname, rcs->path, dirlen);
            strcpy (fullnewname + dirlen, newname);
            free (newname);
            free (rcs->path);
            rcs->path = fullnewname;
        }
    }

    checkin_quiet = flags & RCS_FLAGS_QUIET;
    if (!checkin_quiet)
    {
        cvs_output (rcs->path, 0);
        cvs_output ("  <--  ", 7);
        cvs_output (workfile, 0);
        cvs_output ("\n", 1);
    }

    /* Create new delta node. */
    delta = (RCSVers *) xmalloc (sizeof (RCSVers));
    memset (delta, 0, sizeof (RCSVers));
    delta->author = xstrdup (getcaller ());
    if (flags & RCS_FLAGS_MODTIME)
    {
        struct stat ws;
        if (stat (workfile, &ws) < 0)
        {
            error (1, errno, "cannot stat %s", workfile);
        }
        modtime = ws.st_mtime;
    }
    else
        (void) time (&modtime);
    ftm = gmtime (&modtime);
    delta->date = (char *) xmalloc (MAXDATELEN);
    (void) sprintf (delta->date, DATEFORM,
                    ftm->tm_year + (ftm->tm_year < 100 ? 0 : 1900),
                    ftm->tm_mon + 1, ftm->tm_mday, ftm->tm_hour,
                    ftm->tm_min, ftm->tm_sec);
    if (flags & RCS_FLAGS_DEAD)
    {
        delta->state = xstrdup (RCSDEAD);
        delta->dead = 1;
```

```
    }
5310    else
        delta->state = xstrdup ("Exp");

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* If permissions should be preserved on this project, then
       save the permission info. */
    if (preserve_perms)
    {
        Node *np;
        struct stat sb;
5320    char buf[64]; /* static buffer should be safe: see usage. -twp */

        delta->other_delta = getlist();

        if (CVS_LSTAT (workfile, &sb) < 0)
            error (1, 1, "cannot lstat %s", workfile);

        if (S_ISLNK (sb.st_mode))
        {
            np = getnode();
5330        np->key = xstrdup ("symlink");
            np->data = xreadlink (workfile);
            addnode (delta->other_delta, np);
        }
        else
        {
            (void) sprintf (buf, "%u", sb.st_uid);
            np = getnode();
            np->key = xstrdup ("owner");
            np->data = xstrdup (buf);
5340        addnode (delta->other_delta, np);

            (void) sprintf (buf, "%u", sb.st_gid);
            np = getnode();
            np->key = xstrdup ("group");
            np->data = xstrdup (buf);
            addnode (delta->other_delta, np);

            (void) sprintf (buf, "%o", sb.st_mode & 07777);
            np = getnode();
5350        np->key = xstrdup ("permissions");
            np->data = xstrdup (buf);
            addnode (delta->other_delta, np);

            /* Save device number. */
            switch (sb.st_mode & S_IFMT)
            {
                case S_IFREG: break;
                case S_IFCHR:
                case S_IFBLK:
5360                np = getnode();
                    np->key = xstrdup ("special");
                    sprintf (buf, "%s %lu",
                             ((sb.st_mode & S_IFMT) == S_IFCHR
                              ? "character" : "block"),
                             (unsigned long) sb.st_rdev);
                    np->data = xstrdup (buf);
                    addnode (delta->other_delta, np);
                    break;

5370            default:
                    error (0, 0, "special file %s has unknown type", workfile);
            }

            /* Save hardlinks. */
            delta->hardlinks = list_linked_files_on_disk (workfile);
        }
    }
#endif

5380    /* Create a new deltatext node. */
    dtext = (Deltatext *) xmalloc (sizeof (Deltatext));
    memset (dtext, 0, sizeof (Deltatext));

    dtext->log = make_message_rcslegal (message);

    /* If the delta tree is empty, then there's nothing to link the
       new delta into.  So make a new delta tree, snarf the working
       file contents, and just write the new RCS file. */
    if (rcs->head == NULL)
5390{
        char *newrev;
        FILE *fout;

        /* Figure out what the first revision number should be. */
        if (rev == NULL || *rev == '\0')
            newrev = xstrdup ("1.1");
        else if (numdots (rev) == 0)
        {
```

```
            newrev = (char *) xmalloc (strlen (rev) + 3);
5400        strcpy (newrev, rev);
            strcat (newrev, ".1");
        }
        else
            newrev = xstrdup (rev);

        /* Don't need to xstrdup NEWREV because it's already dynamic, and
           not used for anything else.  (Don't need to free it, either.) */
        rcs->head = newrev;
        delta->version = xstrdup (newrev);
5410    nodep = getnode();
        nodep->type = RCSVERS;
        nodep->key = xstrdup (newrev);
        nodep->data = (char *) delta;
        (void) addnode (rcs->versions, nodep);

        dtext->version = xstrdup (newrev);
        bufsize = 0;
        get_file (workfile, workfile,
                  rcs->expand != NULL && STREQ (rcs->expand, "b") ? "rb" : "r",
5420              &dtext->text, &bufsize, &dtext->len);

        if (!checkin_quiet)
        {
            cvs_output ("initial revision: ", 0);
            cvs_output (rcs->head, 0);
            cvs_output ("\n", 1);
        }

        /* We are probably about to invalidate any cached file.  */
5430    rcsbuf_cache_close ();

        fout = rcs_internal_lockfile (rcs->path);
        RCS_putadmin (rcs, fout);
        RCS_putdtree (rcs, rcs->head, fout);
        RCS_putdesc (rcs, fout);
        rcs->delta_pos = ftell (fout);
        if (rcs->delta_pos == -1)
            error (1, errno, "cannot ftell for %s", rcs->path);
        putdeltatext (fout, dtext);
5440    rcs_internal_unlockfile (fout, rcs->path);
        freedeltatext (dtext);

        if ((flags & RCS_FLAGS_KEEPFILE) == 0)
        {
            if (unlink_file (workfile) < 0)
                /* FIXME-update-dir: message does not include update_dir.  */
                error (0, errno, "cannot remove %s", workfile);
        }

5450    if (!checkin_quiet)
            cvs_output ("done\n", 5);

        return 0;
    }

    /* Derive a new revision number.  From the 'ci' man page:

       "If rev  is  a revision number, it must be higher than the
       latest one on the branch to which  rev  belongs,  or  must
5460   start a new branch.

       If  rev is a branch rather than a revision number, the new
       revision is appended to that branch.  The level number  is
       obtained  by  incrementing the tip revision number of that
       branch.  If rev  indicates  a  non-existing  branch,  that
       branch  is  created  with  the  initial  revision numbered
       rev.1."

      RCS_findlock_or_tip handles the case where REV is omitted.
5470  RCS 5.7 also permits REV to be "$" or to begin with a dot, but
      we do not address those cases - every routine that calls
      RCS_checkin passes it a numeric revision. */

    if (rev == NULL || *rev == '\0')
    {
        /* Figure out where the commit point is by looking for locks.
           If the commit point is at the tip of a branch (or is the
           head of the delta tree), then increment its revision number
           to obtain the new revnum.  Otherwise, start a new
5480       branch. */
        commitpt = RCS_findlock_or_tip (rcs);
        if (commitpt == NULL)
        {
            status = 1;
            goto checkin_done;
        }
        else if (commitpt->next == NULL
                 || STREQ (commitpt->version, rcs->head))
```

```
            delta->version = increment_revnum (commitpt->version);
5490        else
            delta->version = RCS_addbranch (rcs, commitpt->version);
        }
        else
        {
            /* REV is either a revision number or a branch number.  Find the
               tip of the target branch. */
            char *branch, *tip, *newrev, *p;
            int dots, isrevnum;

5500        assert (isdigit(*rev));

            newrev = xstrdup (rev);
            dots = numdots (newrev);
            isrevnum = dots & 1;

            branch = xstrdup (rev);
            if (isrevnum)
            {
                p = strrchr (branch, '.');
5510            *p = '\0';
            }

            /* Find the tip of the target branch.  If we got a one- or two-digit
               revision number, this will be the head of the tree.  Exception:
               if rev is a single-field revision equal to the branch number of
               the trunk (usually "1") then we want to treat it like an ordinary
               branch revision. */
            if (dots == 0)
            {
5520            tip = xstrdup (rcs->head);
                if (atoi (tip) != atoi (branch))
                {
                    newrev = (char *) xrealloc (newrev, strlen (newrev) + 3);
                    strcat (newrev, ".1");
                    dots = isrevnum = 1;
                }
            }
            else if (dots == 1)
                tip = xstrdup (rcs->head);
5530        else
                tip = RCS_getbranch (rcs, branch, 1);

            /* If the branch does not exist, and we were supplied an exact
               revision number, signal an error.  Otherwise, if we were
               given only a branch number, create it and set COMMITPT to
               the branch point. */
            if (tip == NULL)
            {
                if (isrevnum)
5540            {
                    error (0, 0, "%s: can't find branch point %s",
                           rcs->path, branch);
                    free (branch);
                    free (newrev);
                    status = 1;
                    goto checkin_done;
                }
                delta->version = RCS_addbranch (rcs, branch);
                adding_branch = 1;
5550            p = strrchr (branch, '.');
                *p = '\0';
                tip = xstrdup (branch);
            }
            else
            {
                if (isrevnum)
                {
                    /* NEWREV must be higher than TIP. */
                    if (compare_revnums (tip, newrev) >= 0)
5560                {
                        error (0, 0,
                               "%s: revision %s too low; must be higher than %s",
                               rcs->path,
                               newrev, tip);
                        free (branch);
                        free (newrev);
                        free (tip);
                        status = 1;
                        goto checkin_done;
5570                }
                    delta->version = xstrdup (newrev);
                }
                else
                    /* Just increment the tip number to get the new revision. */
                    delta->version = increment_revnum (tip);
            }

            nodep = findnode (rcs->versions, tip);
```

```
    commitpt = (RCSVers *) nodep->data;

    free (branch);
    free (newrev);
    free (tip);
}

assert (delta->version != NULL);

/* If COMMITPT is locked by us, break the lock.  If it's locked
   by someone else, signal an error. */
nodep = findnode (RCS_getlocks (rcs), commitpt->version);
if (nodep != NULL)
{
    if (! STREQ (nodep->data, delta->author))
    {
        /* If we are adding a branch, then leave the old lock around.
           That is sensible in the sense that when adding a branch,
           we don't need to use the lock to tell us where to check
           in.  It is fishy in the sense that if it is our own lock,
           we break it.  However, this is the RCS 5.7 behavior (at
           the end of addbranch in ci.c in RCS 5.7, it calls
           removelock only if it is our own lock, not someone
           else's).  */

        if (!adding_branch)
        {
            error (0, 0, "%s: revision %s locked by %s",
                   rcs->path,
                   nodep->key, nodep->data);
            status = 1;
            goto checkin_done;
        }
    }
    else
        delnode (nodep);
}

dtext->version = xstrdup (delta->version);

/* Obtain the change text for the new delta.  If DELTA is to be the
   new head of the tree, then its change text should be the contents
   of the working file, and LEAFNODE's change text should be a diff.
   Else, DELTA's change text should be a diff between LEAFNODE and
   the working file. */

tmpfile = cvs_temp_name();
status = RCS_checkout (rcs, NULL, commitpt->version, NULL,
                       ((rcs->expand != NULL
                         && STREQ (rcs->expand, "b"))
                        ? "-kb"
                        : "-ko"),
                       tmpfile,
                       (RCSCHECKOUTPROC)0, NULL);
if (status != 0)
    error (1, 0,
           "could not check out revision %s of `%s'",
           commitpt->version, rcs->path);

bufsize = buflen = 0;
chtext = NULL;
chtextlen = 0;
changefile = cvs_temp_name();

/* Diff options should include -binary if the RCS file has -kb set
   in its `expand' field. */
diffopts = (rcs->expand != NULL && STREQ (rcs->expand, "b")
            ? "-a -n --binary"
            : "-a -n");

if (STREQ (commitpt->version, rcs->head) &&
    numdots (delta->version) == 1)
{
    /* If this revision is being inserted on the trunk, the change text
       for the new delta should be the contents of the working file ... */
    bufsize = 0;
    get_file (workfile, workfile,
              rcs->expand != NULL && STREQ (rcs->expand, "b") ? "rb" : "r",
              &dtext->text, &bufsize, &dtext->len);

    /* ... and the change text for the old delta should be a diff. */
    commitpt->text = (Deltatext *) xmalloc (sizeof (Deltatext));
    memset (commitpt->text, 0, sizeof (Deltatext));

    bufsize = 0;
    switch (diff_exec (workfile, tmpfile, diffopts, changefile))
    {
        case 0:
        case 1:
            break;
```

```
            case −1:
5670            /* FIXME-update-dir: message does not include update_dir. */
                error (1, errno, "error diffing %s", workfile);
                break;
            default:
                /* FIXME-update-dir: message does not include update_dir. */
                error (1, 0, "error diffing %s", workfile);
                break;
        }

        /* OK, the text file case here is really dumb. Logically
5680       speaking we want diff to read the files in text mode,
           convert them to the canonical form found in RCS files
           (which, we hope at least, is independent of OS−always
           bare linefeeds), and then work with change texts in that
           format. However, diff_exec both generates change
           texts and produces output for user purposes (e.g. patch.c),
           and there is no way to distinguish between the two cases.
           So we actually implement the text file case by writing the
           change text as a text file, then reading it as a text file.
           This should cause no harm, but doesn't strike me as
5690       immensely clean. */
        get_file (changefile, changefile,
                  rcs−>expand != NULL && STREQ (rcs−>expand, "b") ? "rb" : "r",
                  &commitpt−>text−>text, &bufsize, &commitpt−>text−>len);

        /* If COMMITPT->TEXT->TEXT is NULL, it means that CHANGEFILE
           was empty and that there are no differences between revisions.
           In that event, we want to force RCS_rewrite to write an empty
           string for COMMITPT's change text. Leaving the change text
           field set NULL won't work, since that means "preserve the original
5700       change text for this delta." */
        if (commitpt−>text−>text == NULL)
        {
            commitpt−>text−>text = xstrdup ("");
            commitpt−>text−>len = 0;
        }
    }
    else
    {
        /* This file is not being inserted at the head, but on a side
5710       branch somewhere. Make a diff from the previous revision
           to the working file. */
        switch (diff_exec (tmpfile, workfile, diffopts, changefile))
        {
            case 0:
            case 1:
                break;
            case −1:
                /* FIXME-update-dir: message does not include update_dir. */
                error (1, errno, "error diffing %s", workfile);
5720            break;
            default:
                /* FIXME-update-dir: message does not include update_dir. */
                error (1, 0, "error diffing %s", workfile);
                break;
        }
        /* See the comment above, at the other get_file invocation,
           regarding binary vs. text. */
        get_file (changefile, changefile,
                  rcs−>expand != NULL && STREQ (rcs−>expand, "b") ? "rb" : "r",
5730              &dtext−>text, &bufsize,
                  &dtext−>len);
        if (dtext−>text == NULL)
        {
            dtext−>text = xstrdup ("");
            dtext−>len = 0;
        }
    }

    /* Update DELTA linkage. It is important not to do this before
5740     the very end of RCS_checkin; if an error arises that forces
       us to abort checking in, we must not have malformed deltas
       partially linked into the tree.

       If DELTA and COMMITPT are on different branches, do nothing −
       DELTA is linked to the tree through COMMITPT->BRANCHES, and we
       don't want to change 'next' pointers.

       Otherwise, if the nodes are both on the trunk, link DELTA to
       COMMITPT; otherwise, link COMMITPT to DELTA. */
5750
    if (numdots (commitpt−>version) == numdots (delta−>version))
    {
        if (STREQ (commitpt−>version, rcs−>head))
        {
            delta−>next = rcs−>head;
            rcs−>head = xstrdup (delta−>version);
        }
        else
```

```
            commitpt->next = xstrdup (delta->version);
    }

    /* Add DELTA to RCS->VERSIONS. */
    if (rcs->versions == NULL)
        rcs->versions = getlist();
    nodep = getnode();
    nodep->type = RCSVERS;
    nodep->key = xstrdup (delta->version);
    nodep->data = (char *) delta;
    (void) addnode (rcs->versions, nodep);

    /* Write the new RCS file, inserting the new delta at COMMITPT. */
    if (!checkin_quiet)
    {
        cvs_output ("new revision: ", 14);
        cvs_output (delta->version, 0);
        cvs_output ("; previous revision: ", 21);
        cvs_output (commitpt->version, 0);
        cvs_output ("\n", 1);
    }

    RCS_rewrite (rcs, dtext, commitpt->version);

    if ((flags & RCS_FLAGS_KEEPFILE) == 0)
    {
        if (unlink_file (workfile) < 0)
            /* FIXME-update-dir: message does not include update_dir.  */
            error (1, errno, "cannot remove %s", workfile);
    }
    if (unlink_file (tmpfile) < 0)
        error (0, errno, "cannot remove %s", tmpfile);
    if (unlink_file (changefile) < 0)
        error (0, errno, "cannot remove %s", changefile);

    if (!checkin_quiet)
        cvs_output ("done\n", 5);

 checkin_done:
    if (allocated_workfile)
        free (workfile);

    if (commitpt != NULL && commitpt->text != NULL)
    {
        freedeltatext (commitpt->text);
        commitpt->text = NULL;
    }

    freedeltatext (dtext);
    if (status != 0)
        free_rcsvers_contents (delta);

    return status;
}

/* This structure is passed between RCS_cmp_file and cmp_file_buffer.  */

struct cmp_file_data
{
    const char *filename;
    FILE *fp;
    int different;
};

/* Compare the contents of revision REV of RCS file RCS with the
   contents of the file FILENAME.  OPTIONS is a string for the keyword
   expansion options.  Return 0 if the contents of the revision are
   the same as the contents of the file, 1 if they are different.  */

int
RCS_cmp_file (rcs, rev, options, filename)
     RCSNode *rcs;
     char *rev;
     char *options;
     const char *filename;
{
    int binary;
    FILE *fp;
    struct cmp_file_data data;
    int retcode;

    if (options != NULL && options[0] != '\0')
        binary = STREQ (options, "-kb");
    else
    {
        char *expand;

        expand = RCS_getexpand (rcs);
        if (expand != NULL && STREQ (expand, "b"))
            binary = 1;
```

```
        else
5850        binary = 0;
    }

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* If CVS is to deal properly with special files (when
       PreservePermissions is on), the best way is to check out the
       revision to a temporary file and call 'xcmp' on the two disk
       files.  xcmp needs to handle non-regular files properly anyway,
       so calling it simplifies RCS_cmp_file.  We *could* just yank
       the delta node out of the version tree and look for device
5860   numbers, but writing to disk and calling xcmp is a better
       abstraction (therefore probably more robust). -twp */

    if (preserve_perms)
    {
        char *tmp;

        tmp = cvs_temp_name();
        retcode = RCS_checkout(rcs, NULL, rev, NULL, options, tmp, NULL, NULL);
        if (retcode != 0)
5870        return 1;

        retcode = xcmp (tmp, filename);
        if (CVS_UNLINK (tmp) < 0)
            error (0, errno, "cannot remove %s", tmp);
        return retcode;
    }
    else
#endif
    {
5880    fp = CVS_FOPEN (filename, binary ? FOPEN_BINARY_READ : "r");
        if (fp == NULL)
            /* FIXME-update-dir: should include update_dir in message.  */
            error (1, errno, "cannot open file %s for comparing", filename);

        data.filename = filename;
        data.fp = fp;
        data.different = 0;

        retcode = RCS_checkout (rcs, (char *) NULL, rev, (char *) NULL,
5890                            options, RUN_TTY, cmp_file_buffer,
                                (void *) &data);

        /* If we have not yet found a difference, make sure that we are at
           the end of the file.  */
        if (! data.different)
        {
            if (getc (fp) != EOF)
                data.different = 1;
        }
5900
        fclose (fp);

        if (retcode != 0)
            return 1;

        return data.different;
    }
}

5910 /* This is a subroutine of RCS_cmp_file.  It is passed to
    RCS_checkout.  */

#define CMP_BUF_SIZE (8 * 1024)

static void
cmp_file_buffer (callerdat, buffer, len)
     void *callerdat;
     const char *buffer;
     size_t len;
5920 {
    struct cmp_file_data *data = (struct cmp_file_data *) callerdat;
    char *filebuf;

    /* If we've already found a difference, we don't need to check
       further.  */
    if (data->different)
        return;

    filebuf = xmalloc (len > CMP_BUF_SIZE ? CMP_BUF_SIZE : len);
5930
    while (len > 0)
    {
        size_t checklen;

        checklen = len > CMP_BUF_SIZE ? CMP_BUF_SIZE : len;
        if (fread (filebuf, 1, checklen, data->fp) != checklen)
        {
            if (ferror (data->fp))
```

```
                error (1, errno, "cannot read file %s for comparing",
5940                       data->filename);
                data->different = 1;
                free (filebuf);
                return;
            }

            if (memcmp (filebuf, buffer, checklen) != 0)
            {
                data->different = 1;
                free (filebuf);
5950            return;
            }

            buffer += checklen;
            len -= checklen;
        }

        free (filebuf);
    }

5960 /* For RCS file RCS, make symbolic tag TAG point to revision REV.
       This validates that TAG is OK for a user to use.  Return value is
       -1 for error (and errno is set to indicate the error), positive for
       error (and an error message has been printed), or zero for success.  */

    int
    RCS_settag (rcs, tag, rev)
        RCSNode *rcs;
        const char *tag;
        const char *rev;
5970 {
        List *symbols;
        Node *node;

        if (rcs->flags & PARTIAL)
            RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

        /* FIXME: This check should be moved to RCS_check_tag.  There is no
           reason for it to be here.  */
        if (STREQ (tag, TAG_BASE)
5980        || STREQ (tag, TAG_HEAD))
        {
            /* Print the name of the tag might be considered redundant
               with the caller, which also prints it.  Perhaps this helps
               clarify why the tag name is considered reserved, I don't
               know.  */
            error (0, 0, "Attempt to add reserved tag name %s", tag);
            return 1;
        }

5990    /* A revision number of NULL means use the head or default branch.
           If rev is not NULL, it may be a symbolic tag or branch number;
           expand it to the correct numeric revision or branch head. */
        if (rev == NULL)
            rev = rcs->branch ? rcs->branch : rcs->head;

        /* At this point rcs->symbol_data may not have been parsed.
           Calling RCS_symbols will force it to be parsed into a list
           which we can easily manipulate.  */
        symbols = RCS_symbols (rcs);
6000    if (symbols == NULL)
        {
            symbols = getlist ();
            rcs->symbols = symbols;
        }
        node = findnode (symbols, tag);
        if (node != NULL)
        {
            free (node->data);
            node->data = xstrdup (rev);
6010    }
        else
        {
            node = getnode ();
            node->key = xstrdup (tag);
            node->data = xstrdup (rev);
            (void) addnode_at_front (symbols, node);
        }

        return 0;
6020 }

    /* For RCS file RCS, make symbolic tag TAG point to revision REV.
       This validates that TAG is OK for a user to use.  Return value is
       -1 for error (and errno is set to indicate the error), positive for
       error (and an error message has been printed), or zero for success.  */

    int
    RCS_setremotetag (rcs, tag, rev, remote)
```

```
    RCSNode *rcs;
    const char *tag;
    const char *rev;
    const char* remote;
{
  List* remote_branches;
  Node* node;
  Node* versionnode;
  RCSVers* version;
  char* branchpoint;
  char* p;
  int c = 0;

  /* Set up the association between the tag and the branch as normally */
  int retval = RCS_settag (rcs, tag, rev);
  if (retval != 0) {
    return retval;
  }

  branchpoint = xstrdup (rev);
  for (p = branchpoint + strlen (branchpoint); p > branchpoint; p--) {
    if (*p == '.') {
      c++;
    }
    if (c == 2) {
      *p = '\0';
      break;
    }
  }

  /* Add the branch to the list of remote branches */
  versionnode = findnode (rcs -> versions, branchpoint);
  version = (RCSVers*) (versionnode -> data);
  remote_branches = version -> remote_branches;
  if (remote_branches == NULL) {
    remote_branches = version -> remote_branches = getlist ();
  }

  node = getnode ();
  node -> key = xmalloc (strlen (rev) + strlen (remote) + 3);
  sprintf (node -> key, "%s:%s", rev, remote);
  node -> data = xstrdup ("");

  addnode_at_front (remote_branches, node);
  return 0;
}

/* Delete the symbolic tag TAG from the RCS file RCS.  Return 0 if
   the tag was found (and removed), or 1 if it was not present.  (In
   either case, the tag will no longer be in RCS->SYMBOLS.) */

int
RCS_deltag (rcs, tag)
    RCSNode *rcs;
    const char *tag;
{
    List *symbols;
    Node *node;
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    symbols = RCS_symbols (rcs);
    if (symbols == NULL)
        return 1;

    node = findnode (symbols, tag);
    if (node == NULL)
        return 1;

    delnode (node);

    return 0;
}

/* Set the default branch of RCS to REV.  */

int
RCS_setbranch (rcs, rev)
     RCSNode *rcs;
     const char *rev;
{
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (rev && ! *rev)
        rev = NULL;

    if (rev == NULL && rcs->branch == NULL)
        return 0;
    if (rev != NULL && rcs->branch != NULL && STREQ (rev, rcs->branch))
```

```
        return 0;

    if (rcs->branch != NULL)
        free (rcs->branch);
    rcs->branch = xstrdup (rev);

    return 0;
}

/* Lock revision REV.  LOCK_QUIET is 1 to suppress output.  FIXME:
   Most of the callers only call us because RCS_checkin still tends to
   like a lock (a relic of old behavior inherited from the RCS ci
   program).  If we clean this up, only "cvs admin -l" will still need
   to call RCS_lock.  */

/* FIXME-twp: if a lock owned by someone else is broken, should this
   send mail to the lock owner?  Prompt user?  It seems like such an
   obscure situation for CVS as almost not worth worrying much
   about. */

int
RCS_lock (rcs, rev, lock_quiet)
    RCSNode *rcs;
    const char *rev;
    int lock_quiet;
{
    List *locks;
    Node *p;
    char *user;
    char *xrev = NULL;

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    locks = RCS_getlocks (rcs);
    if (locks == NULL)
        locks = rcs->locks = getlist();
    user = getcaller();

    /* A revision number of NULL means lock the head or default branch. */
    if (rev == NULL)
        xrev = RCS_head (rcs);

    /* If rev is a branch number, lock the latest revision on that
       branch. I think that if the branch doesn't exist, it's
       okay to return 0 - that just means that the branch is new,
       so we don't need to lock it anyway. -twp */
    else if (RCS_nodeisbranch (rcs, rev))
    {
        xrev = RCS_getbranch (rcs, (char *) rev, 1);
        if (xrev == NULL)
        {
            if (!lock_quiet)
                error (0, 0, "%s: branch %s absent", rcs->path, rev);
            return 1;
        }
    }

    if (xrev == NULL)
        xrev = xstrdup (rev);

    /* Make sure that the desired revision exists.  Technically,
       we can update the locks list without even checking this,
       but RCS 5.7 did this.  And it can't hurt. */
    if (findnode (rcs->versions, xrev) == NULL)
    {
        if (!lock_quiet)
            error (0, 0, "%s: revision %s absent", rcs->path, xrev);
        free (xrev);
        return 1;
    }

    /* Is this rev already locked? */
    p = findnode (locks, xrev);
    if (p != NULL)
    {
        if (STREQ (p->data, user))
        {
            /* We already own the lock on this revision, so do nothing. */
            free (xrev);
            return 0;
        }

#if 0
        /* Well, first of all, "rev" below should be "xrev" to avoid
           core dumps.  But more importantly, should we really be
           breaking the lock unconditionally?  What CVS 1.9 does (via
           RCS) is to prompt "Revision 1.1 is already locked by fred.
           Do you want to break the lock? [ny](n): ".  Well, we don't
           want to interact with the user (certainly not at the
```

```
            server/protocol level, and probably not in the command-line
            client), but isn't it more sensible to give an error and
            let the user run "cvs admin -u" if they want to break the
            lock? */

          /* Break the lock. */
          if (!lock_quiet)
          {
              cvs_output (rev, 0);
              cvs_output (" unlocked\n", 0);
          }
          delnode (p);
#else
          error (1, 0, "Revision %s is already locked by %s", xrev, p->data);
#endif
      }

      /* Create a new lock. */
      p = getnode();
      p->key = xrev; /* already xstrdupped */
      p->data = xstrdup (getcaller());
      (void) addnode_at_front (locks, p);

      if (!lock_quiet)
      {
          cvs_output (xrev, 0);
          cvs_output (" locked\n", 0);
      }

      return 0;
}

/* Unlock revision REV.  UNLOCK_QUIET is 1 to suppress output.  FIXME:
   Like RCS_lock, this can become a no-op if we do the checkin
   ourselves.

   If REV is not null and is locked by someone else, break their
   lock and notify them.  It is an open issue whether RCS_unlock
   queries the user about whether or not to break the lock. */

int
RCS_unlock (rcs, rev, unlock_quiet)
     RCSNode *rcs;
     const char *rev;
     int unlock_quiet;
{
    Node *lock;
    List *locks;
    char *user;
    char *xrev = NULL;

    user = getcaller();
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    /* If rev is NULL, unlock the latest revision (first in
       rcs->locks) held by the caller. */
    if (rev == NULL)
    {
        Node *p;

        /* No-ops: attempts to unlock an empty tree or an unlocked file. */
        if (rcs->head == NULL)
        {
            if (!unlock_quiet)
                cvs_outerr ("can't unlock an empty tree\n", 0);
            return 0;
        }

        locks = RCS_getlocks (rcs);
        if (locks == NULL)
        {
            if (!unlock_quiet)
                cvs_outerr ("No locks are set.\n", 0);
            return 0;
        }

        lock = NULL;
        for (p = locks->list->next; p != locks->list; p = p->next)
        {
            if (STREQ (p->data, user))
            {
                if (lock != NULL)
                {
                    if (!unlock_quiet)
                        error (0, 0, "\
%s: multiple revisions locked by %s; please specify one", rcs->path, user);
                    return 1;
                }
                lock = p;
```

```
            }
6300        }
            if (lock == NULL)
                return 0; /* no lock found, ergo nothing to do */
            xrev = xstrdup (lock->key);
        }
        else if (RCS_nodeisbranch (rcs, rev))
        {
            /* If rev is a branch number, unlock the latest revision on that
               branch. */
            xrev = RCS_getbranch (rcs, (char *) rev, 1);
6310        if (xrev == NULL)
            {
                error (0, 0, "%s: branch %s absent", rcs->path, rev);
                return 1;
            }
        }
        else
            /* REV is an exact revision number. */
            xrev = xstrdup (rev);

6320    lock = findnode (RCS_getlocks (rcs), xrev);
        if (lock == NULL)
        {
            /* This revision isn't locked. */
            free (xrev);
            return 0;
        }

        if (! STREQ (lock->data, user))
        {
6330        /* If the revision is locked by someone else, notify
               them.  Note that this shouldn't ever happen if RCS_unlock
               is called with a NULL revision, since that means "whatever
               revision is currently locked by the caller." */
            char *repos, *workfile;
            repos = xstrdup (rcs->path);
            workfile = strrchr (repos, '/');
            *workfile++ = '\0';
            notify_do ('C', workfile, user, NULL, NULL, repos);
            free (repos);
6340    }

        delnode (lock);
        if (!unlock_quiet)
        {
            cvs_output (xrev, 0);
            cvs_output (" unlocked\n", 0);
        }

        free (xrev);
6350    return 0;
    }

    /* Add USER to the access list of RCS.  Do nothing if already present.
       FIXME-twp: check syntax of USER to make sure it's a valid id. */

    void
    RCS_addaccess (rcs, user)
        RCSNode *rcs;
        char *user;
6360 {
        char *access, *a;

        if (rcs->flags & PARTIAL)
            RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

        if (rcs->access == NULL)
            rcs->access = xstrdup (user);
        else
        {
6370        access = xstrdup (rcs->access);
            for (a = strtok (access, " "); a != NULL; a = strtok (NULL, " "))
            {
                if (STREQ (a, user))
                {
                    free (access);
                    return;
                }
            }
            rcs->access = (char *) xrealloc
6380            (rcs->access, strlen (rcs->access) + strlen (user) + 2);
            strcat (rcs->access, " ");
            strcat (rcs->access, user);
        }
    }

    /* Remove USER from the access list of RCS. */

    void
```

```
RCS_delaccess (rcs, user)
    RCSNode *rcs;
    char *user;
{
    char *p, *s;
    int ulen;

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (rcs->access == NULL)
        return;

    p = rcs->access;
    ulen = strlen (user);
    while (p != NULL)
    {
        if (p[ulen] == '\0' || p[ulen] == ' ')
            if (strncmp (p, user, ulen) == 0)
                break;
        p = strchr (p, ' ');
        if (p != NULL)
            ++p;
    }

    if (p == NULL)
        return;

    s = p + ulen;
    while (*s != '\0')
        *p++ = *s++;
    *p = '\0';
}

char *
RCS_getaccess (rcs)
    RCSNode *rcs;
{
    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    return rcs->access;
}

static int findtag PROTO ((Node *, void *));

/* Return a nonzero value if the revision specified by ARG is found.  */

static int
findtag (node, arg)
    Node *node;
    void *arg;
{
    char *rev = (char *)arg;

    if (STREQ (node->data, rev))
        return 1;
    else
        return 0;
}

/* Delete revisions between REV1 and REV2.  The changes between the two
   revisions must be collapsed, and the result stored in the revision
   immediately preceding the lower one.  Return 0 for successful completion,
   1 otherwise.

   Solution: check out the revision preceding REV1 and the revision
   following REV2.  Use call_diff to find aggregate diffs between
   these two revisions, and replace the delta text for the latter one
   with the new aggregate diff.  Alternatively, we could write a
   function that takes two change texts and combines them to produce a
   new change text, without checking out any revs or calling diff.  It
   would be hairy, but so, so cool.

   If INCLUSIVE is set, then TAG1 and TAG2, if non-NULL, tell us to
   delete that revision as well (cvs admin -o tag1:tag2).  If clear,
   delete up to but not including that revision (cvs admin -o tag1::tag2).
   This does not affect TAG1 or TAG2 being NULL; the meaning of the start
   point in ::tag2 and :tag2 is the same and likewise for end points.  */

int
RCS_delete_revs (rcs, tag1, tag2, inclusive)
    RCSNode *rcs;
    char *tag1;
    char *tag2;
    int inclusive;
{
    char *next;
    Node *nodep;
    RCSVers *revp = NULL;
```

```
    RCSVers *beforep;
6480 int status, found;
    int save_noexec;

    char *branchpoint = NULL;
    char *rev1 = NULL;
    char *rev2 = NULL;
    int rev1_inclusive = inclusive;
    int rev2_inclusive = inclusive;
    char *before = NULL;
    char *after = NULL;
6490 char *beforefile = NULL;
    char *afterfile = NULL;
    char *outfile = NULL;

    if (tag1 == NULL && tag2 == NULL)
        return 0;

    /* Assume error status until everything is finished. */
    status = 1;

6500 /* Make sure both revisions exist. */
    if (tag1 != NULL)
    {
        rev1 = RCS_gettag (rcs, tag1, 1, NULL);
        if (rev1 == NULL || (nodep = findnode (rcs->versions, rev1)) == NULL)
        {
            error (0, 0, "%s: Revision %s doesn't exist.", rcs->path, tag1);
            goto delrev_done;
        }
    }
6510 if (tag2 != NULL)
    {
        rev2 = RCS_gettag (rcs, tag2, 1, NULL);
        if (rev2 == NULL || (nodep = findnode (rcs->versions, rev2)) == NULL)
        {
            error (0, 0, "%s: Revision %s doesn't exist.", rcs->path, tag2);
            goto delrev_done;
        }
    }

6520 /* If rev1 is on the trunk and rev2 is NULL, rev2 should be
       RCS->HEAD.  (*Not* RCS_head(rcs), which may return rcs->branch
       instead.)  We need to check this special case early, in order
       to make sure that rev1 and rev2 get ordered correctly. */
    if (rev2 == NULL && numdots (rev1) == 1)
    {
        rev2 = xstrdup (rcs->head);
        rev2_inclusive = 1;
    }

6530 if (rev2 == NULL)
        rev2_inclusive = 1;

    if (rev1 != NULL && rev2 != NULL)
    {
        /* A range consisting of a branch number means the latest revision
           on that branch. */
        if (RCS_isbranch (rcs, rev1) && STREQ (rev1, rev2))
            rev1 = rev2 = RCS_getbranch (rcs, rev1, 0);
        else
6540    {
            /* Make sure REV1 and REV2 are ordered correctly (in the
               same order as the next field).  For revisions on the
               trunk, REV1 should be higher than REV2; for branches,
               REV1 should be lower.  */
            /* Shouldn't we just be giving an error in the case where
               the user specifies the revisions in the wrong order
               (that is, always swap on the trunk, never swap on a
               branch, in the non-error cases)?  It is not at all
               clear to me that users who specify -o 1.4:1.2 really
6550           meant to type -o 1.2:1.4, and the out of order usage
               has never been documented, either by cvs.texinfo or
               rcs(1).  */
            char *temp;
            int temp_inclusive;
            if (numdots (rev1) == 1)
            {
                if (compare_revnums (rev1, rev2) <= 0)
                {
                    temp = rev2;
6560                rev2 = rev1;
                    rev1 = temp;

                    temp_inclusive = rev2_inclusive;
                    rev2_inclusive = rev1_inclusive;
                    rev1_inclusive = temp_inclusive;
                }
            }
            else if (compare_revnums (rev1, rev2) > 0)
```

```
            {
6570            temp = rev2;
                rev2 = rev1;
                rev1 = temp;

                temp_inclusive = rev2_inclusive;
                rev2_inclusive = rev1_inclusive;
                rev1_inclusive = temp_inclusive;
            }
        }
    }
6580
    /* Basically the same thing; make sure that the ordering is what we
       need.  */
    if (rev1 == NULL)
    {
        assert (rev2 != NULL);
        if (numdots (rev2) == 1)
        {
            /* Swap rev1 and rev2.  */
            int temp_inclusive;
6590
            rev1 = rev2;
            rev2 = NULL;

            temp_inclusive = rev2_inclusive;
            rev2_inclusive = rev1_inclusive;
            rev1_inclusive = temp_inclusive;
        }
    }

6600    /* Put the revision number preceding the first one to delete into
       BEFORE (where "preceding" means according to the next field).
       If the first revision to delete is the first revision on its
       branch (e.g. 1.3.2.1), BEFORE should be the node on the trunk
       at which the branch is rooted.  If the first revision to delete
       is the head revision of the trunk, set BEFORE to NULL.

       Note that because BEFORE may not be on the same branch as REV1,
       it is not very handy for navigating the revision tree.  It's
       most useful just for checking out the revision preceding REV1. */
6610    before = NULL;
    branchpoint = RCS_getbranchpoint (rcs, rev1 != NULL ? rev1 : rev2);
    if (rev1 == NULL)
    {
        rev1 = xstrdup (branchpoint);
        if (numdots (branchpoint) > 1)
        {
            char *bp;
            bp = strrchr (branchpoint, '.');
            while (*--bp != '.')
6620            ;
            *bp = '\0';
            /* Note that this is exclusive, always, because the inclusive
               flag doesn't affect the meaning when rev1 == NULL.  */
            before = xstrdup (branchpoint);
            *bp = '.';
        }
    }
    else if (! STREQ (rev1, branchpoint))
    {
6630        /* Walk deltas from BRANCHPOINT on, looking for REV1. */
        nodep = findnode (rcs->versions, branchpoint);
        revp = (RCSVers *) nodep->data;
        while (revp->next != NULL && ! STREQ (revp->next, rev1))
        {
            revp = (RCSVers *) nodep->data;
            nodep = findnode (rcs->versions, revp->next);
        }
        if (revp->next == NULL)
        {
6640        error (0, 0, "%s: Revision %s doesn't exist.", rcs->path, rev1);
            goto delrev_done;
        }
        if (rev1_inclusive)
            before = xstrdup (revp->version);
        else
        {
            before = rev1;
            nodep = findnode (rcs->versions, before);
            rev1 = xstrdup (((RCSVers *)nodep->data)->next);
6650        }
    }
    else if (!rev1_inclusive)
    {
        before = rev1;
        nodep = findnode (rcs->versions, before);
        rev1 = xstrdup (((RCSVers *)nodep->data)->next);
    }
    else if (numdots (branchpoint) > 1)
```

```
    {
6660    /* Example: rev1 is "1.3.2.1", branchpoint is "1.3.2.1".
           Set before to "1.3".  */
        char *bp;
        bp = strrchr (branchpoint, '.');
        while (*--bp != '.')
            ;
        *bp = '\0';
        before = xstrdup (branchpoint);
        *bp = '.';
    }
6670
    /* If any revision between REV1 and REV2 is locked or is a branch point,
       we can't delete that revision and must abort. */
    after = NULL;
    next = rev1;
    found = 0;
    while (!found && next != NULL)
    {
        nodep = findnode (rcs->versions, next);
        revp = (RCSVers *) nodep->data;
6680
        if (rev2 != NULL)
            found = STREQ (revp->version, rev2);
        next = revp->next;

        if ((!found && next != NULL) || rev2_inclusive || rev2 == NULL)
        {
            if (findnode (RCS_getlocks (rcs), revp->version))
            {
                error (0, 0, "%s: can't remove locked revision %s",
6690                   rcs->path,
                       revp->version);
                goto delrev_done;
            }
            if (revp->branches != NULL)
            {
                error (0, 0, "%s: can't remove branch point %s",
                       rcs->path,
                       revp->version);
                goto delrev_done;
6700        }

            /* Doing this only for the :: syntax is for compatibility.
               See cvs.texinfo for somewhat more discussion.  */
            if (!inclusive
                && walklist (RCS_symbols (rcs), findtag, revp->version))
            {
                /* We don't print which file this happens to on the theory
                   that the caller will print the name of the file in a
                   more useful fashion (fullname not rcs->path).  */
6710            error (0, 0, "cannot remove revision %s because it has tags",
                       revp->version);
                goto delrev_done;
            }

            /* It's misleading to print the 'deleting revision' output
               here, since we may not actually delete these revisions.
               But that's how RCS does it.  Bleah.  Someday this should be
               moved to the point where the revs are actually marked for
               deletion. -twp */
6720        cvs_output ("deleting revision ", 0);
            cvs_output (revp->version, 0);
            cvs_output ("\n", 1);
        }
    }

    if (rev2 == NULL)
        ;
    else if (found)
    {
6730    if (rev2_inclusive)
            after = xstrdup (next);
        else
            after = xstrdup (revp->version);
    }
    else if (!inclusive)
    {
        /* In the case of an empty range, for example 1.2::1.2 or
           1.2::1.3, we want to just do nothing.  */
        status = 0;
6740    goto delrev_done;
    }
    else
    {
        /* This looks fishy in the cases where tag1 == NULL or tag2 == NULL.
           Are those cases really impossible?  */
        assert (tag1 != NULL);
        assert (tag2 != NULL);
```

```
        error (0, 0, "%s: invalid revision range %s:%s", rcs->path,
6750            tag1, tag2);
        goto delrev_done;
    }

    if (after == NULL && before == NULL)
    {
        /* The user is trying to delete all revisions.  While an
           RCS file without revisions makes sense to RCS (e.g. the
           state after "rcs -i"), CVS has never been able to cope with
           it.  So at least for now we just make this an error.
6760
           We don't include rcs->path in the message since "cvs admin"
           already printed "RCS file:" and the name.  */
        error (1, 0, "attempt to delete all revisions");
    }

    /* The conditionals at this point get really hairy.  Here is the
       general idea:

       IF before != NULL and after == NULL
6770     THEN don't check out any revisions, just delete them
       IF before == NULL and after != NULL
         THEN only check out after's revision, and use it for the new deltatext
       ELSE
         check out both revisions and diff -n them.  This could use
         RCS_exec_rcsdiff with some changes, like being able
         to suppress diagnostic messages and to direct output. */

    if (after != NULL)
    {
6780    char *diffbuf;
        size_t bufsize, len;

        afterfile = cvs_temp_name();
        status = RCS_checkout (rcs, NULL, after, NULL, NULL, afterfile,
                               (RCSCHECKOUTPROC)0, NULL);
        if (status > 0)
            goto delrev_done;

        if (before == NULL)
6790    {
            /* We are deleting revisions from the head of the tree,
               so must create a new head. */
            diffbuf = NULL;
            bufsize = 0;
            get_file (afterfile, afterfile, "r", &diffbuf, &bufsize, &len);

            save_noexec = noexec;
            noexec = 0;
            if (unlink_file (afterfile) < 0)
6800            error (0, errno, "cannot remove %s", afterfile);
            noexec = save_noexec;

            free (afterfile);
            afterfile = NULL;

            free (rcs->head);
            rcs->head = xstrdup (after);
        }
        else
6810    {
            beforefile = cvs_temp_name();
            status = RCS_checkout (rcs, NULL, before, NULL, NULL, beforefile,
                                   (RCSCHECKOUTPROC)0, NULL);
            if (status > 0)
                goto delrev_done;

            outfile = cvs_temp_name();
            status = diff_exec (beforefile, afterfile, "-n", outfile);

6820        if (status == 2)
            {
                /* Not sure we need this message; will diff_exec already
                   have printed an error?  */
                error (0, 0, "%s: could not diff", rcs->path);
                status = 1;
                goto delrev_done;
            }

            diffbuf = NULL;
6830        bufsize = 0;
            get_file (outfile, outfile, "r", &diffbuf, &bufsize, &len);
        }

        /* Save the new change text in after's delta node. */
        nodep = findnode (rcs->versions, after);
        revp = (RCSVers *) nodep->data;

        assert (revp->text == NULL);
```

```
6840        revp->text = (Deltatext *) xmalloc (sizeof (Deltatext));
            memset ((Deltatext *) revp->text, 0, sizeof (Deltatext));
            revp->text->version = xstrdup (revp->version);
            revp->text->text = diffbuf;
            revp->text->len = len;

            /* If DIFFBUF is NULL, it means that OUTFILE is empty and that
               there are no differences between the two revisions.  In that
               case, we want to force RCS_copydeltas to write an empty string
               for the new change text (leaving the text field set NULL
6850           means "preserve the original change text for this delta," so
               we don't want that). */
            if (revp->text->text == NULL)
                revp->text->text = xstrdup ("");
        }

        /* Walk through the revisions (again) to mark each one as
           outdated.  (FIXME: would it be safe to use the 'dead' field for
           this?  Doubtful.) */
        for (next = rev1;
6860         next != NULL && (after == NULL || ! STREQ (next, after));
             next = revp->next)
        {
            nodep = findnode (rcs->versions, next);
            revp = (RCSVers *) nodep->data;
            revp->outdated = 1;
        }

        /* Update delta links.  If BEFORE == NULL, we're changing the
           head of the tree and don't need to update any 'next' links. */
6870    if (before != NULL)
        {
            /* If REV1 is the first node on its branch, then BEFORE is its
               root node (on the trunk) and we have to update its branches
               list.  Otherwise, BEFORE is on the same branch as AFTER, and
               we can just change BEFORE's 'next' field to point to AFTER.
               (This should be safe: since findnode manages its lists via
               the 'hashnext' and 'hashprev' fields, rather than 'next' and
               'prev', mucking with 'next' and 'prev' should not corrupt the
               delta tree's internal structure.  Much. -twp) */
6880
            if (rev1 == NULL)
                /* beforep's ->next field already should be equal to after,
                   which I think is always NULL in this case.  */
                ;
            else if (STREQ (rev1, branchpoint))
            {
                nodep = findnode (rcs->versions, before);
                revp = (RCSVers *) nodep->data;
                nodep = revp->branches->list->next;
6890            while (nodep != revp->branches->list &&
                       ! STREQ (nodep->key, rev1))
                    nodep = nodep->next;
                assert (nodep != revp->branches->list);
                if (after == NULL)
                    delnode (nodep);
                else
                {
                    free (nodep->key);
                    nodep->key = xstrdup (after);
6900            }
            }
            else
            {
                nodep = findnode (rcs->versions, before);
                beforep = (RCSVers *) nodep->data;
                free (beforep->next);
                beforep->next = xstrdup (after);
            }
        }
6910
        status = 0;

    delrev_done:
        if (rev1 != NULL)
            free (rev1);
        if (rev2 != NULL)
            free (rev2);
        if (branchpoint != NULL)
            free (branchpoint);
6920    if (before != NULL)
            free (before);
        if (after != NULL)
            free (after);

        save_noexec = noexec;
        noexec = 0;
        if (beforefile != NULL)
        {
```

```
        if (unlink_file (beforefile) < 0)
6930        error (0, errno, "cannot remove %s", beforefile);
        free (beforefile);
    }
    if (afterfile != NULL)
    {
        if (unlink_file (afterfile) < 0)
            error (0, errno, "cannot remove %s", afterfile);
        free (afterfile);
    }
    if (outfile != NULL)
6940    {
        if (unlink_file (outfile) < 0)
            error (0, errno, "cannot remove %s", outfile);
        free (outfile);
    }
    noexec = save_noexec;

    return status;
}

6950 /*
 * TRUE if there exists a symbolic tag "tag" in file.
 */
int
RCS_exist_tag (rcs, tag)
    RCSNode *rcs;
    char *tag;
{

    assert (rcs != NULL);
6960
    if (findnode (RCS_symbols (rcs), tag))
    return 1;
    return 0;

}

/*
 * TRUE if RCS revision number "rev" exists.
 * This includes magic branch revisions, not found in rcs->versions,
6970 * but only in rcs->symbols, requiring a list walk to find them.
 * Take advantage of list walk callback function already used by
 * RCS_delete_revs, above.
 */
int
RCS_exist_rev (rcs, rev)
    RCSNode *rcs;
    char *rev;
{

6980    assert (rcs != NULL);

    if (rcs->flags & PARTIAL)
        RCS_reparsercsfile (rcs, (FILE **) NULL, (struct rcsbuffer *) NULL);

    if (findnode(rcs->versions, rev) != 0)
        return 1;

    if (walklist (RCS_symbols(rcs), findtag, rev) != 0)
        return 1;
6990
    return 0;

}


/* RCS_deltas and friends.  Processing of the deltas in RCS files.  */

struct line
{
7000    /* Text of this line.  Part of the same malloc'd block as the struct
       line itself (we probably should use the "struct hack" (char text[1])
       and save ourselves sizeof (char *) bytes).  Does not include \n;
       instead has_newline indicates the presence or absence of \n.  */
    char *text;
    /* Length of this line, not counting \n if has_newline is true.  */
    size_t len;
    /* Version in which it was introduced.  */
    RCSVers *vers;
    /* Nonzero if this line ends with \n.  This will always be true
7010    except possibly for the last line.  */
    int has_newline;
    /* Number of pointers to this struct line.  */
    int refcount;
};

struct linevector
{
    /* How many lines in use for this linevector?  */
```

```
    unsigned int nlines;
7020    /* How many lines allocated for this linevector?  */
    unsigned int lines_alloced;
    /* Pointer to array containing a pointer to each line.  */
    struct line **vector;
};

static void linevector_init PROTO ((struct linevector *));

/* Initialize *VEC to be a linevector with no lines.  */
static void
7030 linevector_init (vec)
    struct linevector *vec;
{
    vec->lines_alloced = 0;
    vec->nlines = 0;
    vec->vector = NULL;
}

static int linevector_add PROTO ((struct linevector *vec, const char *text,
                                  size_t len, RCSVers *vers,
7040                              unsigned int pos));

/* Given some text TEXT, add each of its lines to VEC before line POS
   (where line 0 is the first line).  The last line in TEXT may or may
   not be \n terminated.
   Set the version for each of the new lines to VERS.  This
   function returns non-zero for success.  It returns zero if the line
   number is out of range.

   Each of the lines in TEXT are copied to space which is managed with
7050 the linevector (and freed by linevector_free).  So the caller doesn't
   need to keep TEXT around after the call to this function.  */
static int
linevector_add (vec, text, len, vers, pos)
    struct linevector *vec;
    const char *text;
    size_t len;
    RCSVers *vers;
    unsigned int pos;
{
7060    const char *textend;
    unsigned int i;
    unsigned int nnew;
    const char *p;
    const char *nextline_text;
    size_t nextline_len;
    int nextline_newline;
    struct line *q;

    if (len == 0)
7070    return 1;

    textend = text + len;

    /* Count the number of lines we will need to add.  */
    nnew = 1;
    for (p = text; p < textend; ++p)
    if (*p == '\n' && p + 1 < textend)
        ++nnew;

7080 /* Expand VEC->VECTOR if needed.  */
    if (vec->nlines + nnew >= vec->lines_alloced)
    {
    if (vec->lines_alloced == 0)
        vec->lines_alloced = 10;
    while (vec->nlines + nnew >= vec->lines_alloced)
        vec->lines_alloced *= 2;
    vec->vector = xrealloc (vec->vector,
                            vec->lines_alloced * sizeof (*vec->vector));
    }
7090
    /* Make room for the new lines in VEC->VECTOR.  */
    for (i = vec->nlines + nnew - 1; i >= pos + nnew; --i)
    vec->vector[i] = vec->vector[i - nnew];

    if (pos > vec->nlines)
    return 0;

    /* Actually add the lines, to VEC->VECTOR.  */
    i = pos;
7100 nextline_text = text;
    nextline_newline = 0;
    for (p = text; p < textend; ++p)
    if (*p == '\n')
    {
        nextline_newline = 1;
        if (p + 1 == textend)
        /* If there are no characters beyond the last newline, we
           don't consider it another line.  */
```

```
                break;
7110            nextline_len = p − nextline_text;
                q = (struct line ∗) xmalloc (sizeof (struct line) + nextline_len);
                q−>vers = vers;
                q−>text = (char ∗)q + sizeof (struct line);
                q−>len = nextline_len;
                q−>has_newline = nextline_newline;
                q−>refcount = 1;
                memcpy (q−>text, nextline_text, nextline_len);
                vec−>vector[i++] = q;

7120            nextline_text = (char ∗)p + 1;
                nextline_newline = 0;
            }
        nextline_len = p − nextline_text;
        q = (struct line ∗) xmalloc (sizeof (struct line) + nextline_len);
        q−>vers = vers;
        q−>text = (char ∗)q + sizeof (struct line);
        q−>len = nextline_len;
        q−>has_newline = nextline_newline;
        q−>refcount = 1;
7130    memcpy (q−>text, nextline_text, nextline_len);
        vec−>vector[i] = q;

        vec−>nlines += nnew;

        return 1;
    }

    static void linevector_delete PROTO ((struct linevector ∗, unsigned int,
                                          unsigned int));
7140
    /∗ Remove NLINES lines from VEC at position POS (where line 0 is the
       first line). ∗/
    static void
    linevector_delete (vec, pos, nlines)
        struct linevector ∗vec;
        unsigned int pos;
        unsigned int nlines;
    {
        unsigned int i;
7150    unsigned int last;

        last = vec−>nlines − nlines;
        for (i = pos; i < pos + nlines; ++i)
        {
            if (−−vec−>vector[i]−>refcount == 0)
                free (vec−>vector[i]);
        }
        for (i = pos; i < last; ++i)
            vec−>vector[i] = vec−>vector[i + nlines];
7160    vec−>nlines −= nlines;
    }

    static void linevector_copy PROTO ((struct linevector ∗, struct linevector ∗));

    /∗ Copy FROM to TO, copying the vectors but not the lines pointed to. ∗/
    static void
    linevector_copy (to, from)
        struct linevector ∗to;
        struct linevector ∗from;
7170 {
        unsigned int ln;

        for (ln = 0; ln < to−>nlines; ++ln)
        {
            if (−−to−>vector[ln]−>refcount == 0)
                free (to−>vector[ln]);
        }
        if (from−>nlines > to−>lines_alloced)
        {
7180        if (to−>lines_alloced == 0)
                to−>lines_alloced = 10;
            while (from−>nlines > to−>lines_alloced)
                to−>lines_alloced ∗= 2;
            to−>vector = (struct line ∗∗)
                xrealloc (to−>vector, to−>lines_alloced ∗ sizeof (∗to−>vector));
        }
        memcpy (to−>vector, from−>vector,
                from−>nlines ∗ sizeof (∗to−>vector));
        to−>nlines = from−>nlines;
7190    for (ln = 0; ln < to−>nlines; ++ln)
            ++to−>vector[ln]−>refcount;
    }

    static void linevector_free PROTO ((struct linevector ∗));

    /∗ Free storage associated with linevector. ∗/
    static void
    linevector_free (vec)
```

```
    struct linevector *vec;
7200 {
    unsigned int ln;

    if (vec->vector != NULL)
    {
        for (ln = 0; ln < vec->nlines; ++ln)
            if (--vec->vector[ln]->refcount == 0)
                free (vec->vector[ln]);

        free (vec->vector);
7210    }
}

static char *month_printname PROTO ((char *));

/* Given a textual string giving the month (1-12), terminated with any
   character not recognized by atoi, return the 3 character name to
   print it with.  I do not think it is a good idea to change these
   strings based on the locale; they are standard abbreviations (for
   example in rfc822 mail messages) which should be widely understood.
7220    Returns a pointer into static readonly storage.  */
static char *
month_printname (month)
    char *month;
{
    static const char *const months[] =
      {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
    int mnum;

7230    mnum = atoi (month);
    if (mnum < 1 || mnum > 12)
        return "???";
    return (char *)months[mnum - 1];
}

static int
apply_rcs_changes PROTO ((struct linevector *, const char *, size_t,
                          const char *, RCSVers *, RCSVers *));

7240 /* Apply changes to the line vector LINES.  DIFFBUF is a buffer of
   length DIFFLEN holding the change text from an RCS file (the output
   of diff -n).  NAME is used in error messages.  The VERS field of
   any line added is set to ADDVERS.  The VERS field of any line
   deleted is set to DELVERS, unless DELVERS is NULL, in which case
   the VERS field of deleted lines is unchanged.  The function returns
   non-zero if the change text is applied successfully.  It returns
   zero if the change text does not appear to apply to LINES (e.g., a
   line number is invalid).  If the change text is improperly
   formatted (e.g., it is not the output of diff -n), the function
7250 calls error with a status of 1, causing the program to exit.  */

static int
apply_rcs_changes (lines, diffbuf, difflen, name, addvers, delvers)
     struct linevector *lines;
     const char *diffbuf;
     size_t difflen;
     const char *name;
     RCSVers *addvers;
     RCSVers *delvers;
7260 {
    const char *p;
    const char *q;
    int op;
    /* The RCS format throws us for a loop in that the deltafrags (if
       we define a deltafrag as an add or a delete) need to be applied
       in reverse order.  So we stick them into a linked list.  */
    struct deltafrag {
        enum {ADD, DELETE} type;
        unsigned long pos;
7270    unsigned long nlines;
        const char *new_lines;
        size_t len;
        struct deltafrag *next;
    };
    struct deltafrag *dfhead;
    struct deltafrag *df;

    dfhead = NULL;
    for (p = diffbuf; p != NULL && p < diffbuf + difflen; )
7280    {
        op = *p++;
        if (op != 'a' && op != 'd')
            /* Can't just skip over the deltafrag, because the value
               of op determines the syntax.  */
            error (1, 0, "unrecognized operation '%c' in %s", op, name);
        df = (struct deltafrag *) xmalloc (sizeof (struct deltafrag));
        df->next = dfhead;
        dfhead = df;
```

```
        df->pos = strtoul (p, (char **) &q, 10);
7290
        if (p == q)
            error (1, 0, "number expected in %s", name);
        p = q;
        if (*p++ != ' ')
            error (1, 0, "space expected in %s", name);
        df->nlines = strtoul (p, (char **) &q, 10);
        if (p == q)
            error (1, 0, "number expected in %s", name);
        p = q;
7300    if (*p++ != '\012')
            error (1, 0, "linefeed expected in %s", name);

        if (op == 'a')
        {
            unsigned int i;

            df->type = ADD;
            i = df->nlines;
            /* The text we want is the number of lines specified, or
7310           until the end of the value, whichever comes first (it
               will be the former except in the case where we are
               adding a line which does not end in newline).  */
            for (q = p; i != 0; ++q)
                if (*q == '\n')
                    --i;
                else if (q == diffbuf + difflen)
                {
                    if (i != 1)
                        error (1, 0, "premature end of change in %s", name);
7320                else
                        break;
                }

            /* Stash away a pointer to the text we are adding.  */
            df->new_lines = p;
            df->len = q - p;

            p = q;
        }
7330    else
        {
            /* Correct for the fact that line numbers in RCS files
               start with 1.  */
            --df->pos;

            assert (op == 'd');
            df->type = DELETE;
        }
    }
7340
    for (df = dfhead; df != NULL;)
    {
        unsigned int ln;

        switch (df->type)
        {
        case ADD:
            if (! linevector_add (lines, df->new_lines, df->len, addvers,
                                  df->pos))
7350            return 0;
            break;
        case DELETE:
            if (df->pos > lines->nlines
                || df->pos + df->nlines > lines->nlines)
                return 0;
            if (delvers != NULL)
                for (ln = df->pos; ln < df->pos + df->nlines; ++ln)
                    lines->vector[ln]->vers = delvers;
            linevector_delete (lines, df->pos, df->nlines);
7360        break;
        }
        df = df->next;
        free (dfhead);
        dfhead = df;
    }

    return 1;
}

7370 /* Apply an RCS change text to a buffer.  The function name starts
   with rcs rather than RCS because this does not take an RCSNode
   argument.  NAME is used in error messages.  TEXTBUF is the text
   buffer to change, and TEXTLEN is the size.  DIFFBUF and DIFFLEN are
   the change buffer and size.  The new buffer is returned in *RETBUF
   and *RETLEN.  The new buffer is allocated by xmalloc.

   Return 1 for success.  On failure, call error and return 0.  */
```

```
int
rcs_change_text (name, textbuf, textlen, diffbuf, difflen, retbuf, retlen)
     const char *name;
     char *textbuf;
     size_t textlen;
     const char *diffbuf;
     size_t difflen;
     char **retbuf;
     size_t *retlen;
{
    struct linevector lines;
    int ret;

    *retbuf = NULL;
    *retlen = 0;

    linevector_init (&lines);

    if (! linevector_add (&lines, textbuf, textlen, NULL, 0))
        error (1, 0, "cannot initialize line vector");

    if (! apply_rcs_changes (&lines, diffbuf, difflen, name, NULL, NULL))
    {
        error (0, 0, "invalid change text in %s", name);
        ret = 0;
    }
    else
    {
        char *p;
        size_t n;
        unsigned int ln;

        n = 0;
        for (ln = 0; ln < lines.nlines; ++ln)
            /* 1 for \n */
            n += lines.vector[ln]->len + 1;

        p = xmalloc (n);
        *retbuf = p;

        for (ln = 0; ln < lines.nlines; ++ln)
        {
            memcpy (p, lines.vector[ln]->text, lines.vector[ln]->len);
            p += lines.vector[ln]->len;
            if (lines.vector[ln]->has_newline)
                *p++ = '\n';
        }

        *retlen = p - *retbuf;
        assert (*retlen <= n);

        ret = 1;
    }

    linevector_free (&lines);

    return ret;
}

/* Walk the deltas in RCS to get to revision VERSION.

   If OP is RCS_ANNOTATE, then write annotations using cvs_output.

   If OP is RCS_FETCH, then put the contents of VERSION into a
   newly-malloc'd array and put a pointer to it in *TEXT.  Each line
   is \n terminated; the caller is responsible for converting text
   files if desired.  The total length is put in *LEN.

   If FP is non-NULL, it should be a file descriptor open to the file
   RCS with file position pointing to the deltas.  We close the file
   when we are done.

   If LOG is non-NULL, then *LOG is set to the log message of VERSION,
   and *LOGLEN is set to the length of the log message.

   On error, give a fatal error.  */

static void
RCS_deltas (rcs, fp, rcsbuf, version, op, text, len, log, loglen)
    RCSNode *rcs;
    FILE *fp;
    struct rcsbuffer *rcsbuf;
    char *version;
    enum rcs_delta_op op;
    char **text;
    size_t *len;
    char **log;
    size_t *loglen;
{
    struct rcsbuffer rcsbuf_local;
```

```
    char *branchversion;
7470 char *cpversion;
    char *key;
    char *value;
    size_t vallen;
    RCSVers *vers;
    RCSVers *prev_vers;
    RCSVers *trunk_vers;
    char *next;
    int ishead, isnext, isversion, onbranch;
    Node *node;
7480 struct linevector headlines;
    struct linevector curlines;
    struct linevector trunklines;
    int foundhead;

    if (fp == NULL)
    {
        rcsbuf_cache_open (rcs, rcs->delta_pos, &fp, &rcsbuf_local);
        rcsbuf = &rcsbuf_local;
    }
7490
    ishead = 1;
    vers = NULL;
    prev_vers = NULL;
    trunk_vers = NULL;
    next = NULL;
    onbranch = 0;
    foundhead = 0;

    linevector_init (&curlines);
7500 linevector_init (&headlines);
    linevector_init (&trunklines);

    /* We set BRANCHVERSION to the version we are currently looking
       for.  Initially, this is the version on the trunk from which
       VERSION branches off.  If VERSION is not a branch, then
       BRANCHVERSION is just VERSION.  */
    branchversion = xstrdup (version);
    cpversion = strchr (branchversion, '.');
    if (cpversion != NULL)
7510    cpversion = strchr (cpversion + 1, '.');
    if (cpversion != NULL)
        *cpversion = '\0';

    do {
        if (! rcsbuf_getrevnum (rcsbuf, &key))
            error (1, 0, "unexpected EOF reading RCS file %s", rcs->path);

        if (next != NULL && ! STREQ (next, key))
        {
7520        /* This is not the next version we need.  It is a branch
               version which we want to ignore.  */
            isnext = 0;
            isversion = 0;
        }
        else
        {
            isnext = 1;

            /* look up the revision */
7530        node = findnode (rcs->versions, key);
            if (node == NULL)
                error (1, 0,
                       "mismatch in rcs file %s between deltas and deltatexts",
                       rcs->path);

            /* Stash the previous version.  */
            prev_vers = vers;

            vers = (RCSVers *) node->data;
7540        next = vers->next;

            /* Compare key and trunkversion now, because key points to
               storage controlled by rcsbuf_getkey.  */
            if (STREQ (branchversion, key))
                isversion = 1;
            else
                isversion = 0;
        }

7550    while (1)
        {
            if (! rcsbuf_getkey (rcsbuf, &key, &value))
                error (1, 0, "%s does not appear to be a valid rcs file",
                       rcs->path);

            if (log != NULL
                && isversion
                && STREQ (key, "log")
```

```
            && STREQ (branchversion, version))
7560        {
                *log = rcsbuf_valcopy (rcsbuf, value, 0, loglen);
            }

            if (STREQ (key, "text"))
            {
                rcsbuf_valpolish (rcsbuf, value, 0, &vallen);
                if (ishead)
                {
                    if (! linevector_add (&curlines, value, vallen, NULL, 0))
7570                    error (1, 0, "invalid rcs file %s", rcs->path);

                    ishead = 0;
                }
                else if (isnext)
                {
                    if (! apply_rcs_changes (&curlines, value, vallen,
                                             rcs->path,
                                             onbranch ? vers : NULL,
                                             onbranch ? NULL : prev_vers))
7580                    error (1, 0, "invalid change text in %s", rcs->path);
                }
                break;
            }
        }

        if (isversion)
        {
            /* This is either the version we want, or it is the
               branchpoint to the version we want.  */
7590        if (STREQ (branchversion, version))
            {
                /* This is the version we want.  */
                linevector_copy (&headlines, &curlines);
                foundhead = 1;
                if (onbranch)
                {
                    /* We have found this version by tracking up a
                       branch.  Restore back to the lines we saved
                       when we left the trunk, and continue tracking
7600                   down the trunk.  */
                    onbranch = 0;
                    vers = trunk_vers;
                    next = vers->next;
                    linevector_copy (&curlines, &trunklines);
                }
            }
            else
            {
                Node *p;
7610
                /* We need to look up the branch.  */
                onbranch = 1;

                if (numdots (branchversion) < 2)
                {
                    unsigned int ln;

                    /* We are leaving the trunk; save the current
                       lines so that we can restore them when we
7620                   continue tracking down the trunk.  */
                    trunk_vers = vers;
                    linevector_copy (&trunklines, &curlines);

                    /* Reset the version information we have
                       accumulated so far.  It only applies to the
                       changes from the head to this version.  */
                    for (ln = 0; ln < curlines.nlines; ++ln)
                        curlines.vector[ln]->vers = NULL;
                }
7630
                /* The next version we want is the entry on
                   VERS->branches which matches this branch.  For
                   example, suppose VERSION is 1.21.4.3 and
                   BRANCHVERSION was 1.21.  Then we look for an entry
                   starting with "1.21.4" and we'll put it (probably
                   1.21.4.1) in NEXT.  We'll advance BRANCHVERSION by
                   two dots (in this example, to 1.21.4.3).  */

                if (vers->branches == NULL)
7640                error (1, 0, "missing expected branches in %s",
                           rcs->path);
                *cpversion = '.';
                ++cpversion;
                cpversion = strchr (cpversion, '.');
                if (cpversion == NULL)
                    error (1, 0, "version number confusion in %s",
                           rcs->path);
                for (p = vers->branches->list->next;
```

```
                    p != vers->branches->list;
7650                p = p->next)
                if (strncmp (p->key, branchversion,
                             cpversion - branchversion) == 0)
                    break;
            if (p == vers->branches->list)
                error (1, 0, "missing expected branch in %s",
                       rcs->path);

            next = p->key;

7660        cpversion = strchr (cpversion + 1, '.');
            if (cpversion != NULL)
                *cpversion = '\0';
        }
    }
    if (op == RCS_FETCH && foundhead)
        break;
} while (next != NULL);

free (branchversion);
7670
rcsbuf_cache (rcs, rcsbuf);

if (! foundhead)
    error (1, 0, "could not find desired version %s in %s",
           version, rcs->path);

/* Now print out or return the data we have just computed.  */
switch (op)
{
7680    case RCS_ANNOTATE:
        {
            unsigned int ln;

            for (ln = 0; ln < headlines.nlines; ++ln)
            {
                char buf[80];
                /* Period which separates year from month in date.  */
                char *ym;
                /* Period which separates month from day in date.  */
7690            char *md;
                RCSVers *prvers;

                prvers = headlines.vector[ln]->vers;
                if (prvers == NULL)
                    prvers = vers;

                sprintf (buf, "%-12s (%-8.8s ",
                         prvers->version,
                         prvers->author);
7700            cvs_output (buf, 0);

                /* Now output the date.  */
                ym = strchr (prvers->date, '.');
                if (ym == NULL)
                {
                    /* ??- is an ANSI trigraph.  The ANSI way to
                       avoid it is \? but some pre ANSI compilers
                       complain about the unrecognized escape
                       sequence.  Of course string concatenation
7710                   ("??" "-???") is also an ANSI-ism.  Testing
                       __STDC__ seems to be a can of worms, since
                       compilers do all kinds of things with it.  */
                    cvs_output ("??", 0);
                    cvs_output ("-???", 0);
                    cvs_output ("-??", 0);
                }
                else
                {
                    md = strchr (ym + 1, '.');
7720                if (md == NULL)
                        cvs_output ("??", 0);
                    else
                        cvs_output (md + 1, 2);

                    cvs_output ("-", 1);
                    cvs_output (month_printname (ym + 1), 0);
                    cvs_output ("-", 1);
                    /* Only output the last two digits of the year.  Our output
                       lines are long enough as it is without printing the
7730                   century.  */
                    cvs_output (ym - 2, 2);
                }
                cvs_output ("): ", 0);
                if (headlines.vector[ln]->len != 0)
                    cvs_output (headlines.vector[ln]->text,
                                headlines.vector[ln]->len);
                cvs_output ("\n", 1);
            }
```

```
            }
            break;
        case RCS_FETCH:
            {
                char *p;
                size_t n;
                unsigned int ln;

                assert (text != NULL);
                assert (len != NULL);

                n = 0;
                for (ln = 0; ln < headlines.nlines; ++ln)
                    /* 1 for \n */
                    n += headlines.vector[ln]->len + 1;
                p = xmalloc (n);
                *text = p;
                for (ln = 0; ln < headlines.nlines; ++ln)
                {
                    memcpy (p, headlines.vector[ln]->text,
                            headlines.vector[ln]->len);
                    p += headlines.vector[ln]->len;
                    if (headlines.vector[ln]->has_newline)
                        *p++ = '\n';
                }
                *len = p - *text;
                assert (*len <= n);
            }
            break;
    }

    linevector_free (&curlines);
    linevector_free (&headlines);
    linevector_free (&trunklines);

    return;
}

/* Read the information for a single delta from the RCS buffer RCSBUF,
   whose name is RCSFILE.  *KEYP and *VALP are either NULL, or the
   first key/value pair to read, as set by rcsbuf_getkey. Return NULL
   if there are no more deltas.  Store the key/value pair which
   terminated the read in *KEYP and *VALP.  */

static RCSVers *
getdelta (rcsbuf, rcsfile, keyp, valp)
    struct rcsbuffer *rcsbuf;
    char *rcsfile;
    char **keyp;
    char **valp;
{
    RCSVers *vnode;
    char *key, *value, *keybuf, *valbuf, *cp;
    Node *kv;

    /* Get revision number if it wasn't passed in. This uses
       rcsbuf_getkey because it doesn't croak when encountering
       unexpected input.  As a result, we have to play unholy games
       with 'key' and 'value'. */
    if (*keyp != NULL)
    {
        key = *keyp;
        value = *valp;
    }
    else
    {
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
            error (1, 0, "%s: unexpected EOF", rcsfile);
    }

    /* Make sure that it is a revision number and not a cabbage
       or something. */
    for (cp = key; (isdigit (*cp) || *cp == '.') && *cp != '\0'; cp++)
        /* do nothing */ ;
    /* Note that when comparing with RCSDATE, we are not massaging
       VALUE from the string found in the RCS file.  This is OK since
       we know exactly what to expect.  */
    if (*cp != '\0' || strncmp (RCSDATE, value, (sizeof RCSDATE) - 1) != 0)
    {
        *keyp = key;
        *valp = value;
        return NULL;
    }

    vnode = (RCSVers *) xmalloc (sizeof (RCSVers));
    memset (vnode, 0, sizeof (RCSVers));

    vnode->version = xstrdup (key);

    /* Grab the value of the date from value.  Note that we are not
```

```
        massaging VALUE from the string found in the RCS file. */
7830    cp = value + (sizeof RCSDATE) - 1; /* skip the "date" keyword */
        while (whitespace (*cp))      /* take space off front of value */
            cp++;

        vnode->date = xstrdup (cp);

        /* Get author field. */
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
            error (1, 0, "unexpected end of file reading %s", rcsfile);
7840    }
        if (! STREQ (key, "author"))
            error (1, 0, "\
unable to parse %s; `author' not in the expected place", rcsfile);
        vnode->author = rcsbuf_valcopy (rcsbuf, value, 0, (size_t *) NULL);

        /* Get state field. */
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
            error (1, 0, "unexpected end of file reading %s", rcsfile);
7850    }
        if (! STREQ (key, "state"))
            error (1, 0, "\
unable to parse %s; `state' not in the expected place", rcsfile);
        vnode->state = rcsbuf_valcopy (rcsbuf, value, 0, (size_t *) NULL);
        /* The value is optional, according to rcsfile(5). */
        if (value != NULL && STREQ (value, "dead"))
        {
            vnode->dead = 1;
        }
7860
        /* Note that "branches" and "next" are in fact mandatory, according
           to doc/RCSFILES. */

        /* fill in the branch list (if any branches exist) */
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
            error (1, 0, "unexpected end of file reading %s", rcsfile);
        }
        if (STREQ (key, RCSDESC))
7870    {
            *keyp = key;
            *valp = value;
            /* Probably could/should be a fatal error. */
            error (0, 0, "warning: 'branches' keyword missing from %s", rcsfile);
            return vnode;
        }
        if (value != (char *) NULL)
        {
            vnode->branches = getlist ();
7880        /* Note that we are not massaging VALUE from the string found
               in the RCS file. */
            do_branches (vnode->branches, value);
        }

        /* fill in the next field if there is a next revision */
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
            error (1, 0, "unexpected end of file reading %s", rcsfile);
        }
7890    if (STREQ (key, RCSDESC))
        {
            *keyp = key;
            *valp = value;
            /* Probably could/should be a fatal error. */
            error (0, 0, "warning: 'next' keyword missing from %s", rcsfile);
            return vnode;
        }
        if (value != (char *) NULL)
            vnode->next = rcsbuf_valcopy (rcsbuf, value, 0, (size_t *) NULL);
7900
        /* read the "remote branches" list (optional) */

        if (! rcsbuf_getkey (rcsbuf, &key, &value))
        {
            error (1, 0, "unexpected end of file reading %s", rcsfile);
        }
        if (STREQ (key, "remote-branches"))
        {
            vnode->remote_branches = getlist ();
7910        /* Note that we are not massaging VALUE from the string found
               in the RCS file. */
            do_remote_branches (vnode->remote_branches, value);
        }

        /*
         * XXX - this is where we put the symbolic link stuff???
         * (into newphrases in the deltas).
         */
```

```
    while (1)
7920    {
            int len;
            size_t valbuflen;

            key = NULL;

            if (! rcsbuf_getid (rcsbuf, &keybuf))
                error (1, 0, "unexpected end of file reading %s", rcsfile);

            /* rcsbuf_getid did not terminate the key, so copy it to new space. */
7930        len = rcsbuf->ptr - keybuf;
            key = (char *) xmalloc (sizeof(char) * (len + 1));
            strncpy (key, keybuf, len);
            key[len] = '\0';

            /* The 'desc' keyword has only a single string value, with no
               trailing semicolon, so it must be handled specially. */
            if (STREQ (key, RCSDESC))
            {
                (void) rcsbuf_getstring (rcsbuf, &valbuf);
7940            value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen);
                break;
            }

#ifdef PRESERVE_PERMISSIONS_SUPPORT
            /* The 'hardlinks' value is a group of words, which must
               be parsed separately and added as a list to vnode->hardlinks. */
            if (STREQ (key, "hardlinks"))
            {
                Node *n;
7950
                vnode->hardlinks = getlist();
                while (1)
                {
                    if (! rcsbuf_getword (rcsbuf, &valbuf))
                        error (1, 0, "unexpected end of file reading %s", rcsfile);
                    if (valbuf == NULL)
                        break;
                    n = getnode();
                    n->key = rcsbuf_valcopy (rcsbuf, valbuf, 1, NULL);
7960                addnode (vnode->hardlinks, n);
                }
                continue;
            }
#endif

            /* Get the value. */
            value = NULL;
            while (1)
            {
7970            if (! rcsbuf_getword (rcsbuf, &valbuf))
                    error (1, 0, "unexpected end of file reading %s", rcsfile);
                if (valbuf == NULL)
                    break;

                /* Copy valbuf to new space so we can polish it, then
                   append it to value. */

                if (value == NULL)
                {
7980                value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen);
                }
                else
                {
                    char *temp_value;

                    temp_value = rcsbuf_valcopy (rcsbuf, valbuf, 1, &valbuflen);
                    len = strlen (value);
                    value = (char *) xrealloc
                        (value, sizeof(char) * (len + valbuflen + 2));
7990                value[len] = ' ';
                    strcpy (value + len + 1, temp_value);
                    free (temp_value);
                }
            }

            /* Enable use of repositories created by certain obsolete
               versions of CVS.  This code should remain indefinately;
               there is no procedure for converting old repositories, and
               checking for it is harmless.  */
8000        if (STREQ (key, RCSDEAD))
            {
                vnode->dead = 1;
                if (vnode->state != NULL)
                    free (vnode->state);
                vnode->state = xstrdup ("dead");
                continue;
            }
            /* if we have a new revision number, we're done with this delta */
```

```
        for (cp = key; (isdigit (*cp) || *cp == '.') && *cp != '\0'; cp++)
8010        /* do nothing */ ;
        /* Note that when comparing with RCSDATE, we are not massaging
           VALUE from the string found in the RCS file.  This is OK
           since we know exactly what to expect.  */
        if (*cp == '\0' && strncmp (RCSDATE, value, strlen (RCSDATE)) == 0)
            break;

        /* At this point, key and value represent a user-defined field
           in the delta node. */
        if (vnode->other_delta == NULL)
8020        vnode->other_delta = getlist ();
        kv = getnode ();
        kv->type = RCSFIELD;
        kv->key = key;
        kv->data = value;
        if (addnode (vnode->other_delta, kv) != 0)
        {
            /* Complaining about duplicate keys in newphrases seems
               questionable, in that we don't know what they mean and
               doc/RCSFILES has no prohibition on several newphrases
8030           with the same key.  But we can't store more than one as
               long as we store them in a List *.  */
            error (0, 0, "warning: duplicate key `%s' in RCS file `%s'",
                   key, rcsfile);
            freenode (kv);
        }
    }

    /* Return the key which caused us to fail back to the caller.  */
    *keyp = key;
8040    *valp = value;

    return vnode;
}

static void
freedeltatext (d)
    Deltatext *d;
{
    if (d->version != NULL)
8050    free (d->version);
    if (d->log != NULL)
        free (d->log);
    if (d->text != NULL)
        free (d->text);
    if (d->other != (List *) NULL)
        dellist (&d->other);
    free (d);
}

8060 static Deltatext *
RCS_getdeltatext (rcs, fp, rcsbuf)
    RCSNode *rcs;
    FILE *fp;
    struct rcsbuffer *rcsbuf;
{
    char *num;
    char *key, *value;
    Node *p;
    Deltatext *d;
8070
    /* Get the revision number. */
    if (! rcsbuf_getrevnum (rcsbuf, &num))
    {
        /* If num == NULL, it means we reached EOF naturally.  That's
           fine. */
        if (num == NULL)
            return NULL;
        else
            error (1, 0, "%s: unexpected EOF", rcs->path);
8080    }

    p = findnode (rcs->versions, num);
    if (p == NULL)
        error (1, 0, "mismatch in rcs file %s between deltas and deltatexts",
               rcs->path);

    d = (Deltatext *) xmalloc (sizeof (Deltatext));
    d->version = xstrdup (num);

8090 /* Get the log message. */
    if (! rcsbuf_getkey (rcsbuf, &key, &value))
        error (1, 0, "%s, delta %s: unexpected EOF", rcs->path, num);
    if (! STREQ (key, "log"))
        error (1, 0, "%s, delta %s: expected `log', got `%s'",
               rcs->path, num, key);
    d->log = rcsbuf_valcopy (rcsbuf, value, 0, (size_t *) NULL);

    /* Get random newphrases. */
```

```
    d->other = getlist();
8100 while (1)
    {
        if (! rcsbuf_getkey (rcsbuf, &key, &value))
            error (1, 0, "%s, delta %s: unexpected EOF", rcs->path, num);

        if (STREQ (key, "text"))
            break;

        p = getnode();
        p->type = RCSFIELD;
8110    p->key = xstrdup (key);
        p->data = rcsbuf_valcopy (rcsbuf, value, 1, (size_t *) NULL);
        if (addnode (d->other, p) < 0)
        {
            error (0, 0, "warning: %s, delta %s: duplicate field `%s'",
                   rcs->path, num, key);
        }
    }

    /* Get the change text. We already know that this key is `text'. */
8120 d->text = rcsbuf_valcopy (rcsbuf, value, 0, &d->len);

    return d;
}

/* RCS output functions, for writing RCS format files from RCSNode
   structures.

   For most of this work, RCS 5.7 uses an `aprintf' function which aborts
   program upon error.  Instead, these functions check the output status
8130 of the stream right before closing it, and aborts if an error condition
   is found.  The RCS solution is probably the better one: it produces
   more overhead, but will produce a clearer diagnostic in the case of
   catastrophic error.  In either case, however, the repository will probably
   not get corrupted. */

static int
putsymbol_proc (symnode, fparg)
    Node *symnode;
    void *fparg;
8140 {
    FILE *fp = (FILE *) fparg;

    /* A fiddly optimization: this code used to just call fprintf, but
       in an old repository with hundreds of tags this can get called
       hundreds of thousands of times when doing a cvs tag.  Since
       tagging is a relatively common operation, and using putc and
       fputs is just as comprehensible, the change is worthwhile.  */
    putc ('\n', fp);
    putc ('\t', fp);
8150 fputs (symnode->key, fp);
    putc (':', fp);
    fputs (symnode->data, fp);
    return 0;
}

static int putlock_proc PROTO ((Node *, void *));

/* putlock_proc is like putsymbol_proc, but key and data are reversed. */

8160 static int
putlock_proc (symnode, fp)
    Node *symnode;
    void *fp;
{
    return fprintf ((FILE *) fp, "\n\t%s:%s", symnode->data, symnode->key);
}

static int
putrcsfield_proc (node, vfp)
8170 Node *node;
    void *vfp;
{
    FILE *fp = (FILE *) vfp;

    /* Some magic keys used internally by CVS start with ';'. Skip them. */
    if (node->key[0] == ';')
        return 0;

    fprintf (fp, "\n%s\t", node->key);
8180 if (node->data != NULL)
    {
        /* If the field's value contains evil characters,
           it must be stringified. */
        /* FIXME: This does not quite get it right.  "7jk8f" is not a legal
           value for a value in a newpharse, according to doc/RCSFILES,
           because digits are not valid in an "id".  We might do OK by
           always writing strings (enclosed in @).  Would be nice to
           explicitly mention this one way or another in doc/RCSFILES.
```

```
            A case where we are wrong in a much more clear-cut way is that
8190        we let through non-graphic characters such as whitespace and
            control characters.  */
          int n = strcspn (node->data, "$,.:;@");
          if (node->data[n] == 0)
            fputs (node->data, fp);
          else
          {
            putc ('@', fp);
            expand_at_signs (node->data, (off_t) strlen (node->data), fp);
            putc ('@', fp);
8200      }
        }

        /* desc, log and text fields should not be terminated with semicolon;
           all other fields should be. */
        if (! STREQ (node->key, "desc") &&
            ! STREQ (node->key, "log") &&
            ! STREQ (node->key, "text"))
        {
          putc (';', fp);
8210    }
        return 0;
      }

      #ifdef PRESERVE_PERMISSIONS_SUPPORT

      /* Save a filename in a 'hardlinks' RCS field.  NODE->KEY will contain
         a full pathname, but currently only basenames are stored in the RCS
         node.  Assume that the filename includes nasty characters and
         */
8220
      static int
      puthardlink_proc (node, vfp)
          Node *node;
          void *vfp;
      {
          FILE *fp = (FILE *) vfp;
          char *basename = strrchr (node->key, '/');

          if (basename == NULL)
8230        basename = node->key;
          else
            ++basename;

          putc ('\t', fp);
          putc ('@', fp);
          (void) expand_at_signs (basename, strlen (basename), fp);
          putc ('@', fp);

          return 0;
8240  }

      #endif

      /* Output the admin node for RCS into stream FP. */

      static void
      RCS_putadmin (rcs, fp)
          RCSNode *rcs;
          FILE *fp;
8250  {
          fprintf (fp, "%s\t%s;\n", RCSHEAD, rcs->head ? rcs->head : "");
          if (rcs->branch)
            fprintf (fp, "%s\t%s;\n", RCSBRANCH, rcs->branch);

          fputs ("access", fp);
          if (rcs->access)
          {
            char *p, *s;
            s = xstrdup (rcs->access);
8260        for (p = strtok (s, " \n\t"); p != NULL; p = strtok (NULL, " \n\t"))
              fprintf (fp, "\n\t%s", p);
            free (s);
          }
          fputs (";\n", fp);

          fputs (RCSSYMBOLS, fp);
          /* If we haven't had to convert the symbols to a list yet, don't
             force a conversion now; just write out the string.  */
          if (rcs->symbols == NULL && rcs->symbols_data != NULL)
8270      {
            fputs ("\n\t", fp);
            fputs (rcs->symbols_data, fp);
          }
          else
            walklist (RCS_symbols (rcs), putsymbol_proc, (void *) fp);
          fputs (";\n", fp);

          fputs ("locks", fp);
```

```
    if (rcs->locks_data)
8280    fprintf (fp, "\t%s", rcs->locks_data);
    else if (rcs->locks)
        walklist (rcs->locks, putlock_proc, (void *) fp);
    if (rcs->strict_locks)
        fprintf (fp, "; strict");
    fputs (";\n", fp);

    if (rcs->comment)
    {
        fprintf (fp, "comment\t@");
8290    expand_at_signs (rcs->comment, (off_t) strlen (rcs->comment), fp);
        fputs ("@;\n", fp);
    }
    if (rcs->expand && ! STREQ (rcs->expand, "kv"))
        fprintf (fp, "%s\t@%s@;\n", RCSEXPAND, rcs->expand);

    walklist (rcs->other, putrcsfield_proc, (void *) fp);

    putc ('\n', fp);
}
8300
static void
putdelta (vers, fp)
    RCSVers *vers;
    FILE *fp;
{
    Node *bp, *start;

    /* Skip if no revision was supplied, or if it is outdated (cvs admin -o) */
    if (vers == NULL || vers->outdated)
8310    return;

    fprintf (fp, "\n%s\n%s\t%s;\t%s %s;\t%s %s;\nbranches",
             vers->version,
             RCSDATE, vers->date,
             "author", vers->author,
             "state", vers->state ? vers->state : "");

    if (vers->branches != NULL)
    {
8320    start = vers->branches->list;
        for (bp = start->next; bp != start; bp = bp->next)
            fprintf (fp, "\n\t%s", bp->key);
    }

    fprintf (fp, ";\nnext\t%s;", vers->next ? vers->next : "");

    fprintf (fp, "\nremote-branches");

    if (vers->remote_branches != NULL)
8330    {
      start = vers->remote_branches->list;
      for (bp = start->next; bp != start; bp = bp->next)
        fprintf (fp, "\n\t%s", bp->key);
    }
    fprintf (fp, ";\n");

    walklist (vers->other_delta, putrcsfield_proc, fp);

#ifdef PRESERVE_PERMISSIONS_SUPPORT
8340    if (vers->hardlinks)
    {
        fprintf (fp, "\nhardlinks");
        walklist (vers->hardlinks, puthardlink_proc, fp);
        putc (';', fp);
    }
#endif
    putc ('\n', fp);
}

8350 static void
RCS_putdtree (rcs, rev, fp)
    RCSNode *rcs;
    char *rev;
    FILE *fp;
{
    RCSVers *versp;
    Node *p, *branch;

    if (rev == NULL)
8360    return;

    /* Find the delta node for this revision. */
    p = findnode (rcs->versions, rev);
    assert (p != NULL);
    versp = (RCSVers *) p->data;

    /* Print the delta node and recurse on its 'next' node.  This prints
       the trunk.  If there are any branches printed on this revision,
```

```
        print those trunks as well. */
8370    putdelta (versp, fp);
    RCS_putdtree (rcs, versp->next, fp);
    if (versp->branches != NULL)
    {
        branch = versp->branches->list;
        for (p = branch->next; p != branch; p = p->next)
            RCS_putdtree (rcs, p->key, fp);
    }
}

8380 static void
RCS_putdesc (rcs, fp)
    RCSNode *rcs;
    FILE *fp;
{
    fprintf (fp, "\n\n%s\n@", RCSDESC);
    if (rcs->desc != NULL)
    {
        off_t len = (off_t) strlen (rcs->desc);
        if (len > 0)
8390    {
            expand_at_signs (rcs->desc, len, fp);
            if (rcs->desc[len-1] != '\n')
                putc ('\n', fp);
        }
    }
    fputs ("@\n", fp);
}

    static void
8400 putdeltatext (fp, d)
    FILE *fp;
    Deltatext *d;
{
    fprintf (fp, "\n\n%s\nlog\n@", d->version);
    if (d->log != NULL)
    {
        int loglen = strlen (d->log);
        expand_at_signs (d->log, (off_t) loglen, fp);
        if (d->log[loglen-1] != '\n')
8410        putc ('\n', fp);
    }
    putc ('@', fp);

    walklist (d->other, putrcsfield_proc, fp);

    fputs ("\ntext\n@", fp);
    if (d->text != NULL)
        expand_at_signs (d->text, (off_t) d->len, fp);
    fputs ("@\n", fp);
8420 }

/* TODO: the whole mechanism for updating deltas is kludgey... more
   sensible would be to supply all the necessary info in a 'newdeltatext'
   field for RCSVers nodes. -twp */

/* Copy delta text nodes from FIN to FOUT.  If NEWDTEXT is non-NULL, it
   is a new delta text node, and should be added to the tree at the
   node whose revision number is INSERTPT.  (Note that trunk nodes are
   written in decreasing order, and branch nodes are written in
8430 increasing order.) */

static void
RCS_copydeltas (rcs, fin, rcsbufin, fout, newdtext, insertpt)
    RCSNode *rcs;
    FILE *fin;
    struct rcsbuffer *rcsbufin;
    FILE *fout;
    Deltatext *newdtext;
    char *insertpt;
8440 {
    int actions;
    RCSVers *dadmin;
    Node *np;
    int insertbefore, found;
    char *bufrest;
    int nls;
    size_t buflen;
    char buf[8192];
    int got;
8450
    /* Count the number of versions for which we have to do some
       special operation.  */
    actions = walklist (rcs->versions, count_delta_actions, (void *) NULL);

    /* Make a note of whether NEWDTEXT should be inserted
       before or after its INSERTPT. */
    insertbefore = (newdtext != NULL && numdots (newdtext->version) == 1);
```

```
    while (actions != 0 || newdtext != NULL)
8460 {
        Deltatext *dtext;

        dtext = RCS_getdeltatext (rcs, fin, rcsbufin);

        /* We shouldn't hit EOF here, because that would imply that
           some action was not taken, or that we could not insert
           NEWDTEXT.  */
        if (dtext == NULL)
            error (1, 0, "internal error: EOF too early in RCS_copydeltas");
8470
        found = (insertpt != NULL && STREQ (dtext->version, insertpt));
        if (found && insertbefore)
        {
            putdeltatext (fout, newdtext);
            newdtext = NULL;
            insertpt = NULL;
        }

        np = findnode (rcs->versions, dtext->version);
8480    dadmin = (RCSVers *) np->data;

        /* If this revision has been outdated, just skip it. */
        if (dadmin->outdated)
        {
            --actions;
            continue;
        }

        /* Update the change text for this delta.  New change text
8490       data may come from cvs admin -m, cvs admin -o, or cvs ci. */
        if (dadmin->text != NULL)
        {
            if (dadmin->text->log != NULL || dadmin->text->text != NULL)
                --actions;
            if (dadmin->text->log != NULL)
            {
                free (dtext->log);
                dtext->log = dadmin->text->log;
                dadmin->text->log = NULL;
8500        }
            if (dadmin->text->text != NULL)
            {
                free (dtext->text);
                dtext->text = dadmin->text->text;
                dtext->len = dadmin->text->len;
                dadmin->text->text = NULL;
            }
        }
        putdeltatext (fout, dtext);
8510    freedeltatext (dtext);

        if (found && !insertbefore)
        {
            putdeltatext (fout, newdtext);
            newdtext = NULL;
            insertpt = NULL;
        }
    }

8520 /* Copy the rest of the file directly, without bothering to
       interpret it.  The caller will handle error checking by calling
       ferror.

       We just wrote a newline to the file, either in putdeltatext or
       in the caller.  However, we may not have read the corresponding
       newline from the file, because rcsbuf_getkey returns as soon as
       it finds the end of the '' string for the desc or text key.
       Therefore, we may read three newlines when we should really
       only write two, and we check for that case here.  This is not
8530   an semantically important issue; we only do it to make our RCS
       files look traditional.  */

    nls = 3;

    rcsbuf_get_buffered (rcsbufin, &bufrest, &buflen);
    if (buflen > 0)
    {
        if (bufrest[0] != '\n'
            || strncmp (bufrest, "\n\n\n", buflen < 3 ? buflen : 3) != 0)
8540    {
            nls = 0;
        }
        else
        {
            if (buflen < 3)
                nls -= buflen;
            else
            {
```

```
            ++bufrest;
            --buflen;
            nls = 0;
          }
       }

       fwrite (bufrest, 1, buflen, fout);
    }

    while ((got = fread (buf, 1, sizeof buf, fin)) != 0)
    {
       if (nls > 0
          && got >= nls
          && buf[0] == '\n'
          && strncmp (buf, "\n\n\n", nls) == 0)
       {
          fwrite (buf + 1, 1, got - 1, fout);
       }
       else
       {
          fwrite (buf, 1, got, fout);
       }

       nls = 0;
    }
}

/* A helper procedure for RCS_copydeltas.  This is called via walklist
   to count the number of RCS revisions for which some special action
   is required.  */

static int
count_delta_actions (np, ignore)
   Node *np;
   void *ignore;
{
   RCSVers *dadmin;

   dadmin = (RCSVers *) np->data;

   if (dadmin->outdated)
      return 1;

   if (dadmin->text != NULL
      && (dadmin->text->log != NULL || dadmin->text->text != NULL))
   {
      return 1;
   }

   return 0;
}

/* RCS_internal_lockfile and RCS_internal_unlockfile perform RCS-style
   locking on the specified RCSFILE: for a file called `foo,v', open
   for writing a file called ',foo,'.

   Note that we what do here is quite different from what RCS does.
   RCS creates the ,foo, file before it reads the RCS file (if it
   knows that it will be writing later), so that it actually serves as
   a lock.  We don't; instead we rely on CVS writelocks.  This means
   that if someone is running RCS on the file at the same time they
   are running CVS on it, they might lose (we read the file,
   then RCS writes it, then we write it, clobbering the
   changes made by RCS).  I believe the current sentiment about this
   is "well, don't do that".

   A concern has been expressed about whether adopting the RCS
   strategy would slow us down.  I don't think so, since we need to
   write the ,foo, file anyway (unless perhaps if O_EXCL is slower or
   something).

   These do not perform quite the same function as the RCS -l option
   for locking files: they are intended to prevent competing RCS
   processes from stomping all over each other's laundry.  Hence,
   they are 'internal' locking functions.

   Note that we don't clean up the ,foo, file on ^C.  We probably should.
   I'm not completely sure whether RCS does or not (I looked at the code
   a little, and didn't find it).

   If there is an error, give a fatal error; if we return we always
*/

static FILE *
rcs_internal_lockfile (rcsfile)
   char *rcsfile;
{
   char *lockfile;
   int fd;
   struct stat rstat;
```

```
    FILE *fp;

    /* Get the lock file name: ',file,' for RCS file 'file,v'. */
    lockfile = rcs_lockfilename (rcsfile);

    /* Use the existing RCS file mode, or read-only if this is a new
       file.  (Really, this is a lie – if this is a new file,
       RCS_checkin uses the permissions from the working copy.  For
       actually creating the file, we use 0444 as a safe default mode.) */
    if (stat (rcsfile, &rstat) < 0)
    {
        if (existence_error (errno))
            rstat.st_mode = S_IRUSR | S_IRGRP | S_IROTH;
        else
            error (1, errno, "cannot stat %s", rcsfile);
    }

    /* Try to open exclusively.  POSIX.1 guarantees that O_EXCL|O_CREAT
       guarantees an exclusive open.  According to the RCS source, with
       NFS v2 we must also throw in O_TRUNC and use an open mask that makes
       the file unwriteable.  For extensive justification, see the comments for
       rcswriteopen() in rcsedit.c, in RCS 5.7.  This is kind of pointless
       in the CVS case; see comment at the start of this file concerning
       general ,foo, file strategy.

       There is some sentiment that with NFSv3 and such, that one can
       rely on O_EXCL these days.  This might be true for unix (I
       don't really know), but I am still pretty skeptical in the case
       of the non-unix systems.  */
    fd = open (lockfile, OPEN_BINARY | O_WRONLY | O_CREAT | O_EXCL | O_TRUNC,
               S_IRUSR | S_IRGRP | S_IROTH);

    if (fd < 0)
    {
        error (1, errno, "could not open lock file `%s'", lockfile);
    }

    /* Force the file permissions, and return a stream object. */
    /* Because we change the modes later, we don't worry about
       this in the non-HAVE_FCHMOD case.  */
#ifdef HAVE_FCHMOD
    if (fchmod (fd, rstat.st_mode) < 0)
        error (1, errno, "cannot change mode for %s", lockfile);
#endif
    fp = fdopen (fd, FOPEN_BINARY_WRITE);
    if (fp == NULL)
        error (1, errno, "cannot fdopen %s", lockfile);

    free (lockfile);

    return fp;
}

static void
rcs_internal_unlockfile (fp, rcsfile)
    FILE *fp;
    char *rcsfile;
{
    char *lockfile;

    /* Get the lock file name: ',file,' for RCS file 'file,v'. */
    lockfile = rcs_lockfilename (rcsfile);

    /* Abort if we could not write everything successfully to LOCKFILE.
       This is not a great error-handling mechanism, but should prevent
       corrupting the repository. */

    if (ferror (fp))
        /* The only case in which using errno here would be meaningful
           is if we happen to have left errno unmolested since the call
           which produced the error (e.g. fprintf).  That is pretty
           fragile even if it happens to sometimes be true.  The real
           solution is to check each call to fprintf rather than waiting
           until the end like this.  */
        error (1, 0, "error writing to lock file %s", lockfile);
    if (fclose (fp) == EOF)
        error (1, errno, "error closing lock file %s", lockfile);

    rename_file (lockfile, rcsfile);
    free (lockfile);
}

static char *
rcs_lockfilename (rcsfile)
    char *rcsfile;
{
    char *lockfile, *lockp;
    char *rcsbase, *rcsp, *rcsend;
    int rcslen;
```

```
    /* Create the lockfile name. */
8730    rcslen = strlen (rcsfile);
    lockfile = (char *) xmalloc (rcslen + 10);
    rcsbase = last_component (rcsfile);
    rcsend = rcsfile + rcslen - sizeof(RCSEXT);
    for (lockp = lockfile, rcsp = rcsfile; rcsp < rcsbase; ++rcsp)
        *lockp++ = *rcsp;
    *lockp++ = ',';
    while (rcsp <= rcsend)
        *lockp++ = *rcsp++;
    *lockp++ = ',';
8740    *lockp = '\0';

    return lockfile;
}

/* Rewrite an RCS file.  The basic idea here is that the caller should
   first call RCS_reparsercsfile, then munge the data structures as
   desired (via RCS_delete_revs, RCS_settag, &c), then call RCS_rewrite.  */

void
8750 RCS_rewrite (rcs, newdtext, insertpt)
    RCSNode *rcs;
    Deltatext *newdtext;
    char *insertpt;
{
    FILE *fin, *fout;
    struct rcsbuffer rcsbufin;

    if (noexec)
        return;
8760
    fout = rcs_internal_lockfile (rcs->path);

    RCS_putadmin (rcs, fout);
    RCS_putdtree (rcs, rcs->head, fout);
    RCS_putdesc (rcs, fout);

    /* Open the original RCS file and seek to the first delta text. */
    rcsbuf_cache_open (rcs, rcs->delta_pos, &fin, &rcsbufin);

8770    /* Update delta_pos to the current position in the output file.
       Do NOT move these statements: they must be done after fin has
       been positioned at the old delta_pos, but before any delta
       texts have been written to fout. */
    rcs->delta_pos = ftell (fout);
    if (rcs->delta_pos == -1)
        error (1, errno, "cannot ftell in RCS file %s", rcs->path);

    RCS_copydeltas (rcs, fin, &rcsbufin, fout, newdtext, insertpt);

8780    /* We don't want to call rcsbuf_cache here, since we're about to
       delete the file.  */
    rcsbuf_close (&rcsbufin);
    if (ferror (fin))
        /* The only case in which using errno here would be meaningful
           is if we happen to have left errno unmolested since the call
           which produced the error (e.g. fread).  That is pretty
           fragile even if it happens to sometimes be true.  The real
           solution is to make sure that all the code which reads
           from fin checks for errors itself (some does, some doesn't).  */
8790        error (0, 0, "warning: when closing RCS file `%s'", rcs->path);
    if (fclose (fin) < 0)
        error (0, errno, "warning: closing RCS file `%s'", rcs->path);

    rcs_internal_unlockfile (fout, rcs->path);
}


/* Annotate command.  In rcs.c for historical reasons (from back when
   what is now RCS_deltas was part of annotate_fileproc).  */
8800
/* Options from the command line.  */

static int force_tag_match = 1;
static char *tag = NULL;
static char *date = NULL;

static int annotate_fileproc PROTO ((void *callerdat, struct file_info *));

static int
8810 annotate_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    FILE *fp = NULL;
    struct rcsbuffer *rcsbufp = NULL;
    struct rcsbuffer rcsbuf;
    char *version;
```

```
    if (finfo->rcs == NULL)
8820        return (1);

    if (finfo->rcs->flags & PARTIAL)
    {
        RCS_reparsercsfile (finfo->rcs, &fp, &rcsbuf);
        rcsbufp = &rcsbuf;
    }

    version = RCS_getversion (finfo->rcs, tag, date, force_tag_match,
                              (int *) NULL);
8830    if (version == NULL)
        return 0;

    /* Distinguish output for various files if we are processing
       several files.  */
    cvs_outerr ("Annotations for ", 0);
    cvs_outerr (finfo->fullname, 0);
    cvs_outerr ("\n***************\n", 0);

    RCS_deltas (finfo->rcs, fp, rcsbufp, version, RCS_ANNOTATE, (char **) NULL,
8840            (size_t) NULL, (char **) NULL, (size_t *) NULL);
    free (version);
    return 0;
}

static const char *const annotate_usage[] =
{
    "Usage: %s %s [-lRf] [-r rev|-D date] [files. . .]\n",
    "\t-l\tLocal directory only, no recursion.\n",
    "\t-R\tProcess directories recursively.\n",
8850    "\t-f\tUse head revision if tag/date not found.\n",
    "\t-r rev\tAnnotate file as of specified revision/tag.\n",
    "\t-D date\tAnnotate file as of specified date.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

/* Command to show the revision, date, and author where each line of a
   file was modified.  */

8860 int
annotate (argc, argv)
    int argc;
    char **argv;
{
    int local = 0;
    int c;

    if (argc == -1)
        usage (annotate_usage);
8870
    optind = 0;
    while ((c = getopt (argc, argv, "+lr:D:fR")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
8880            local = 0;
                break;
            case 'r':
                tag = optarg;
                break;
            case 'D':
                date = Make_Date (optarg);
                break;
            case 'f':
                force_tag_match = 0;
8890            break;
            case '?':
            default:
                usage (annotate_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

8900 #ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();

        if (local)
            send_arg ("-l");
        if (!force_tag_match)
```

```
            send_arg ("-f");
        option_with_arg ("-r", tag);
        if (date)
            client_senddate (date);
        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
        send_to_server ("annotate\012", 0);
        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    if (tag != NULL)
        tag_check_valid (tag, argc, argv, local, 0, "");

    return start_recursion (annotate_fileproc, (FILESDONEPROC) NULL,
                            (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                            argc, argv, local, W_LOCAL, 0, 1, (char *)NULL,
                            1);
}

/*
 * For a given file with full pathname PATH and revision number REV,
 * produce a file label suitable for passing to diff.  The default
 * file label as used by RCS 5.7 looks like this:
 *
 *     FILENAME <tab> YYYY/MM/DD <sp> HH:MM:SS <tab> REVNUM
 *
 * The date and time used are the revision's last checkin date and time.
 * If REV is NULL, use the working copy's mtime instead.
 */
char *
make_file_label (path, rev, rcs)
    char *path;
    char *rev;
    RCSNode *rcs;
{
    char datebuf[MAXDATELEN];
    char *label;
    char *file;

    file = last_component (path);
    label = (char *) xmalloc (strlen (file)
                              + (rev == NULL ? 0 : strlen (rev))
                              + 50);

    if (rev)
    {
        char *date;
        RCS_getrevtime (rcs, rev, datebuf, 0);
        date = printable_date (datebuf);
        (void) sprintf (label, "-L%s\t%s\t%s", file, date, rev);
        free (date);
    }
    else
    {
        struct stat sb;
        struct tm *wm;

        if (CVS_STAT (file, &sb) < 0)
            error (0, 1, "could not get info for `%s'", path);
        else
        {
            wm = gmtime (&sb.st_mtime);
            (void) sprintf (datebuf, "%04d/%02d/%02d %02d:%02d:%02d",
                            wm->tm_year + 1900, wm->tm_mon + 1,
                            wm->tm_mday, wm->tm_hour,
                            wm->tm_min, wm->tm_sec);
            (void) sprintf (label, "-L%s\t%s", file, datebuf);
        }
    }
    return label;
}
```

A.45 rcs.h

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * RCS source control definitions needed by rcs.c and friends
 */

/* String which indicates a conflict if it occurs at the start of a line. */
#define RCS_MERGE_PAT ">>>>>>> "

#define RCSEXT ",v"
#define RCSPAT "*,v"
#define RCSHEAD "head"
#define RCSBRANCH "branch"
#define RCSSYMBOLS "symbols"
#define RCSDATE "date"
#define RCSDESC "desc"
#define RCSEXPAND "expand"
#define RCS_REMOTE_BRANCH "1.1.3"

/* Used by the version of death support which resulted from old
   versions of CVS (e.g. 1.5 if you define DEATH_SUPPORT and not
   DEATH_STATE).  Only a hacked up RCS (used by those old versions of
   CVS) will put this into RCS files.  Considered obsolete. */
#define RCSDEAD "dead"

#define DATEFORM "%02d.%02d.%02d.%02d.%02d.%02d"
#define SDATEFORM "%d.%d.%d.%d.%d.%d"

/*
 * Opaque structure definitions used by RCS specific lookup routines
 */
#define VALID 0x1               /* flags field contains valid data */
#define INATTIC 0x2             /* RCS file is located in the Attic */
#define PARTIAL 0x4             /* RCS file not completly parsed */

/* All the "char *" fields in RCSNode, Deltatext, and RCSVers are
   '\0'-terminated (except "text" in Deltatext).  This means that we
   can't deal with fields containing '\0', which is a limitation that
   RCS does not have.  Would be nice to fix this some day. */

struct rcsnode
{
    /* Reference count for this structure.  Used to deal with the
       fact that there might be a pointer from the Vers_TS or might
       not.  Callers who increment this field are responsible for
       calling freercsnode when they are done with their reference. */
    int refcount;

    /* Flags (INATTIC, PARTIAL, &c), see above. */
    int flags;

    /* File name of the RCS file.  This is not necessarily the name
       as specified by the user, but it is a name which can be passed to
       system calls and a name which is OK to print in error messages
       (the various names might differ in case). */
    char *path;

    /* Value for head keyword from RCS header, or NULL if empty. */
    char *head;

    /* Value for branch keyword from RCS header, or NULL if omitted. */
    char *branch;

    /* Raw data on symbolic revisions.  The first time that RCS_symbols is
       called, we parse these into ->symbols, and free ->symbols_data. */
    char *symbols_data;

    /* Value for expand keyword from RCS header, or NULL if omitted. */
    char *expand;

    /* List of nodes, the key of which is the symbolic name and the data
       of which is the numeric revision that it corresponds to (malloc'd). */
    List *symbols;

    /* List of nodes (type RCSVERS), the key of which the numeric revision
       number, and the data of which is an RCSVers * for the revision. */
    List *versions;

    /* Value for access keyword from RCS header, or NULL if empty.
       FIXME: RCS_delaccess would also seem to use "" for empty.  We
       should pick one or the other. */
    char *access;

    /* Raw data on locked revisions.  The first time that RCS_getlocks is
```

```
        called, we parse these into ->locks, and free ->locks_data.  */
90      char *locks_data;

        /* List of nodes, the key of which is the numeric revision and the
           data of which is the user that it corresponds to (malloc'd).  */
        List *locks;

        /* Set for the strict keyword from the RCS header.  */
        int strict_locks;

        /* Value for the comment keyword from RCS header (comment leader), or
100        NULL if omitted.  */
        char *comment;

        /* Value for the desc field in the RCS file, or NULL if empty.  */
        char *desc;

        /* File offset of the first deltatext node, so we can seek there.  */
        long delta_pos;

        /* Newphrases from the RCS header.  List of nodes, the key of which
110        is the "id" which introduces the newphrase, and the value of which
           is the value from the newphrase.  */
        List *other;
    };

    typedef struct rcsnode RCSNode;

    struct deltatext {
        char *version;

120     /* Log message, or NULL if we do not intend to change the log message
           (that is, RCS_copydeltas should just use the log message from the
           file).  */
        char *log;

        /* Change text, or NULL if we do not intend to change the change text
           (that is, RCS_copydeltas should just use the change text from the
           file).  Note that it is perfectly legal to have log be NULL and
           text non-NULL, or vice-versa.  */
        char *text;
130     size_t len;

        /* Newphrase fields from deltatext nodes.  FIXME: duplicates the
           other field in the rcsversnode, I think.  */
        List *other;
    };
    typedef struct deltatext Deltatext;

    struct rcsversnode
    {
140     /* Duplicate of the key by which this structure is indexed.  */
        char *version;

        char *date;
        char *author;
        char *state;
        char *next;
        int dead;
        int outdated;
        Deltatext *text;
150     List *branches;
        List *remote_branches;
        /* Newphrase fields from deltatext nodes.  Also contains ";add" and
           ";delete" magic fields (see rcs.c, log.c).  I think this is
           only used by log.c (where it looks up "log").  Duplicates the
           other field in struct deltatext, I think.  */
        List *other;
        /* Newphrase fields from delta nodes.  */
        List *other_delta;
    #ifdef PRESERVE_PERMISSIONS_SUPPORT
160     /* Hard link information for each revision. */
        List *hardlinks;
    #endif
    };
    typedef struct rcsversnode RCSVers;

    /*
     * CVS reserves all even-numbered branches for its own use.  "magic" branches
     * (see rcs.c) are contained as virtual revision numbers (within symbolic
     * tags only) off the RCS_MAGIC_BRANCH, which is 0.  CVS also reserves the
170  * ".1" branch for vendor revisions.  So, if you do your own branching, you
     * should limit your use to odd branch numbers starting at 3.
     */
    #define RCS_MAGIC_BRANCH 0

    /* The type of a function passed to RCS_checkout.  */
    typedef void (*RCSCHECKOUTPROC) PROTO ((void *, const char *, size_t));

    #ifdef __STDC__
```

```
struct rcsbuffer;
#endif

/*
 * exported interfaces
 */
RCSNode *RCS_parse PROTO((const char *file, const char *repos));
RCSNode *RCS_parsercsfile PROTO((char *rcsfile));
void RCS_fully_parse PROTO((RCSNode *));
void RCS_reparsercsfile PROTO((RCSNode *, FILE **, struct rcsbuffer *));

char *RCS_check_kflag PROTO((const char *arg));
char *RCS_getdate PROTO((RCSNode * rcs, char *date, int force_tag_match));
char *RCS_gettag PROTO((RCSNode * rcs, char *symtag, int force_tag_match,
                        int *simple_tag));
int RCS_exist_rev PROTO((RCSNode *rcs, char *rev));
int RCS_exist_tag PROTO((RCSNode *rcs, char *tag));
char *RCS_tag2rev PROTO((RCSNode *rcs, char *tag));
char *RCS_getversion PROTO((RCSNode * rcs, char *tag, char *date,
                        int force_tag_match, int *simple_tag));
char *RCS_getremoteversion PROTO((struct file_info* finfo, RCSNode * rcs, char *tag, char** local_tag));
char *RCS_magicrev PROTO((RCSNode *rcs, char *rev));
int RCS_isbranch PROTO((RCSNode *rcs, const char *rev));
int RCS_nodeisbranch PROTO((RCSNode *rcs, const char *tag));
char *RCS_whatbranch PROTO((RCSNode *rcs, const char *tag));
char *RCS_head PROTO((RCSNode * rcs));
int RCS_datecmp PROTO((char *date1, char *date2));
time_t RCS_getrevtime PROTO((RCSNode * rcs, char *rev, char *date, int fudge));
List *RCS_symbols PROTO((RCSNode *rcs));
void RCS_check_tag PROTO((const char *tag));
int RCS_valid_rev PROTO ((char *rev));
List *RCS_getlocks PROTO((RCSNode *rcs));
void freercsnode PROTO((RCSNode ** rnodep));
char *RCS_getbranch PROTO((RCSNode * rcs, char *tag, int force_tag_match));
char *RCS_branch_head PROTO ((RCSNode *rcs, char *rev));

int RCS_isdead PROTO((RCSNode *, const char *));
char *RCS_getexpand PROTO ((RCSNode *));
int RCS_checkout PROTO ((RCSNode *, char *, char *, char *, char *, char *,
                         RCSCHECKOUTPROC, void *));
int RCS_checkin PROTO ((RCSNode *rcs, char *workfile, char *message,
                        char *rev, int flags));
int RCS_cmp_file PROTO ((RCSNode *, char *, char *, const char *));
int RCS_settag PROTO ((RCSNode *, const char *, const char *));
int RCS_deltag PROTO ((RCSNode *, const char *));
int RCS_setbranch PROTO((RCSNode *, const char *));
int RCS_lock PROTO ((RCSNode *, const char *, int));
int RCS_unlock PROTO ((RCSNode *, const char *, int));
int RCS_delete_revs PROTO ((RCSNode *, char *, char *, int));
void RCS_addaccess PROTO ((RCSNode *, char *));
void RCS_delaccess PROTO ((RCSNode *, char *));
char *RCS_getaccess PROTO ((RCSNode *));
void RCS_rewrite PROTO ((RCSNode *, Deltatext *, char *));
int rcs_change_text PROTO ((const char *, char *, size_t, const char *,
                            size_t, char **, size_t *));
char *make_file_label PROTO ((char *, char *, RCSNode *));

extern int preserve_perms;

/* From import.c.  */
extern int add_rcs_file PROTO ((char *, char *, char *, char *, char *,
                                char *, char *, int, char **,
                                char *, size_t, char *, FILE *));
```

A.46 rcscmds.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * The functions in this file provide an interface for performing
 * operations directly on RCS files.
 */

#include "cvs.h"
#include <assert.h>
#include <stdio.h>
#include "diffrun.h"

/* This file, rcs.h, and rcs.c, together sometimes known as the "RCS
   library", are intended to define our interface to RCS files.

   Whether there will also be a version of RCS which uses this
   library, or whether the library will be packaged for uses beyond
   CVS or RCS (many people would like such a thing) is an open
   question.  Some considerations:

   1.  An RCS library for CVS must have the capabilities of the
   existing CVS code which accesses RCS files.  In particular, simple
   approaches will often be slow.

   2.  An RCS library should not use code from the current RCS
   (5.7 and its ancestors).  The code has many problems.  Too few
   comments, too many layers of abstraction, too many global variables
   (the correct number for a library is zero), too much intricately
   interwoven functionality, and too many clever hacks.  Paul Eggert,
   the current RCS maintainer, agrees.

   3.  More work needs to be done in terms of separating out the RCS
   library from the rest of CVS (for example, cvs_output should be
   replaced by a callback, and the declarations should be centralized
   into rcs.h, and probably other such cleanups).

   4.  To be useful for RCS and perhaps for other uses, the library
   may need features beyond those needed by CVS.

   5.  Any changes to the RCS file format *must* be compatible.  Many,
   many tools (not just CVS and RCS) can at least import this format.
   RCS and CVS must preserve the current ability to import/export it
   (preferably improved–magic branches are currently a roadblock).
   See doc/RCSFILES in the CVS distribution for documentation of this
   file format.

   On a related note, see the comments at diff_exec, later in this file,
   for more on the diff library.  */

static void RCS_output_diff_options PROTO ((char *, char *, char *, char *));


/* Stuff to deal with passing arguments the way libdiff.a wants to deal
   with them.  This is a crufty interface; there is no good reason for it
   to resemble a command line rather than something closer to "struct
   log_data" in log.c.  */

/* First call call_diff_setup to setup any initial arguments.  The
   argument will be parsed into whitespace separated words and added
   to the global call_diff_argv list.

   Then, optionally, call call_diff_arg for each additional argument
   that you'd like to pass to the diff library.

   Finally, call call_diff or call_diff3 to produce the diffs.  */

static char **call_diff_argv;
static int call_diff_argc;
static int call_diff_argc_allocated;

static void call_diff_add_arg PROTO ((const char *));
static void call_diff_setup PROTO ((const char *prog));
static int call_diff PROTO ((char *out));
static int call_diff3 PROTO ((char *out));

static void call_diff_write_output PROTO((const char *, size_t));
static void call_diff_flush_output PROTO((void));
static void call_diff_write_stdout PROTO((const char *));
static void call_diff_error PROTO((const char *, const char *, const char *));

/* VARARGS */
static void
call_diff_setup (prog)
    const char *prog;
```

```
{
    char *cp;
    int i;
    char *call_diff_prog;

    /* clean out any malloc'ed values from call_diff_argv */
    for (i = 0; i < call_diff_argc; i++)
    {
        if (call_diff_argv[i])
        {
            free (call_diff_argv[i]);
            call_diff_argv[i] = (char *) 0;
        }
    }
    call_diff_argc = 0;

    call_diff_prog = xstrdup (prog);

    /* put each word into call_diff_argv, allocating it as we go */
    for (cp = strtok (call_diff_prog, " \t");
         cp != NULL;
         cp = strtok ((char *) NULL, " \t"))
        call_diff_add_arg (cp);
    free (call_diff_prog);
}

static void
call_diff_arg (s)
    const char *s;
{
    call_diff_add_arg (s);
}

static void
call_diff_add_arg (s)
    const char *s;
{
    /* allocate more argv entries if we've run out */
    if (call_diff_argc >= call_diff_argc_allocated)
    {
        call_diff_argc_allocated += 50;
        call_diff_argv = (char **)
            xrealloc ((char *) call_diff_argv,
                      call_diff_argc_allocated * sizeof (char **));
    }

    if (s)
        call_diff_argv[call_diff_argc++] = xstrdup (s);
    else
        /* Not post-incremented on purpose!  */
        call_diff_argv[call_diff_argc] = (char *) 0;
}

/* Callback function for the diff library to write data to the output
   file.  This is used when we are producing output to stdout.  */

static void
call_diff_write_output (text, len)
    const char *text;
    size_t len;
{
    cvs_output (text, len);
}

/* Call back function for the diff library to flush the output file.
   This is used when we are producing output to stdout.  */

static void
call_diff_flush_output ()
{
    cvs_flushout ();
}

/* Call back function for the diff library to write to stdout.  */

static void
call_diff_write_stdout (text)
    const char *text;
{
    cvs_output (text, 0);
}

/* Call back function for the diff library to write to stderr.  */

static void
call_diff_error (format, a1, a2)
    const char *format;
    const char *a1;
    const char *a2;
{
```

```
    /* FIXME: Should we somehow indicate that this error is coming from
       the diff library?  */
    error (0, 0, format, a1, a2);
}

/* This set of callback functions is used if we are sending the diff
   to stdout.  */

static struct diff_callbacks call_diff_stdout_callbacks =
{
    call_diff_write_output,
    call_diff_flush_output,
    call_diff_write_stdout,
    call_diff_error
};

/* This set of callback functions is used if we are sending the diff
   to a file.  */

static struct diff_callbacks call_diff_file_callbacks =
{
    (void (*) PROTO((const char *, size_t))) NULL,
    (void (*) PROTO((void))) NULL,
    call_diff_write_stdout,
    call_diff_error
};

static int
call_diff (out)
    char *out;
{
    if (out == RUN_TTY)
        return diff_run (call_diff_argc, call_diff_argv, NULL,
                         &call_diff_stdout_callbacks);
    else
        return diff_run (call_diff_argc, call_diff_argv, out,
                         &call_diff_file_callbacks);
}

static int
call_diff3 (out)
    char *out;
{
    if (out == RUN_TTY)
        return diff3_run (call_diff_argc, call_diff_argv, NULL,
                          &call_diff_stdout_callbacks);
    else
        return diff3_run (call_diff_argc, call_diff_argv, out,
                          &call_diff_file_callbacks);
}



/* Merge revisions REV1 and REV2. */

int
RCS_merge(rcs, path, workfile, options, rev1, rev2)
    RCSNode *rcs;
    char *path;
    char *workfile;
    char *options;
    char *rev1;
    char *rev2;
{
    char *xrev1, *xrev2;
    char *tmp1, *tmp2;
    char *diffout = NULL;
    int retval;

    if (options != NULL && options[0] != '\0')
      assert (options[0] == '-' && options[1] == 'k');

    cvs_output ("RCS file: ", 0);
    cvs_output (rcs->path, 0);
    cvs_output ("\n", 1);

    /* Calculate numeric revision numbers from rev1 and rev2 (may be
       symbolic). */
    xrev1 = RCS_gettag (rcs, rev1, 0, NULL);
    xrev2 = RCS_gettag (rcs, rev2, 0, NULL);

    /* Check out chosen revisions.  The error message when RCS_checkout
       fails is not very informative - it is taken verbatim from RCS 5.7,
       and relies on RCS_checkout saying something intelligent upon failure. */
    cvs_output ("retrieving revision ", 0);
    cvs_output (xrev1, 0);
    cvs_output ("\n", 1);

    tmp1 = cvs_temp_name();
    if (RCS_checkout (rcs, NULL, xrev1, rev1, options, tmp1,
```

```
                    (RCSCHECKOUTPROC)0, NULL))
    {
        cvs_outerr ("rcsmerge: co failed\n", 0);
        error_exit();
    }

    cvs_output ("retrieving revision ", 0);
    cvs_output (xrev2, 0);
    cvs_output ("\n", 1);

    tmp2 = cvs_temp_name();
    if (RCS_checkout (rcs, NULL, xrev2, rev2, options, tmp2,
                    (RCSCHECKOUTPROC)0, NULL))
    {
        cvs_outerr ("rcsmerge: co failed\n", 0);
        error_exit();
    }

    /* Merge changes. */
    cvs_output ("Merging differences between ", 0);
    cvs_output (xrev1, 0);
    cvs_output (" and ", 0);
    cvs_output (xrev2, 0);
    cvs_output (" into ", 0);
    cvs_output (workfile, 0);
    cvs_output ("\n", 1);

    /* Remember that the first word in the 'call_diff_setup' string is used now
       only for diagnostic messages - CVS no longer forks to run diff3. */
    diffout = cvs_temp_name();
    call_diff_setup ("diff3");
    call_diff_arg ("-E");
    call_diff_arg ("-am");

    call_diff_arg ("-L");
    call_diff_arg (workfile);
    call_diff_arg ("-L");
    call_diff_arg (xrev1);
    call_diff_arg ("-L");
    call_diff_arg (xrev2);

    call_diff_arg (workfile);
    call_diff_arg (tmp1);
    call_diff_arg (tmp2);

    retval = call_diff3 (diffout);

    if (retval == 1)
        cvs_outerr ("rcsmerge: warning: conflicts during merge\n", 0);
    else if (retval == 2)
        error_exit();

    if (diffout)
        copy_file (diffout, workfile);

    /* Clean up. */
    {
        int save_noexec = noexec;
        noexec = 0;
        if (unlink_file (tmp1) < 0)
        {
            if (!existence_error (errno))
                error (0, errno, "cannot remove temp file %s", tmp1);
        }
        free (tmp1);
        if (unlink_file (tmp2) < 0)
        {
            if (!existence_error (errno))
                error (0, errno, "cannot remove temp file %s", tmp2);
        }
        free (tmp2);
        if (diffout)
        {
            if (unlink_file (diffout) < 0)
            {
                if (!existence_error (errno))
                    error (0, errno, "cannot remove temp file %s", diffout);
            }
            free (diffout);
        }
        free (xrev1);
        free (xrev2);
        noexec = save_noexec;
    }

    return retval;
}

/* Diff revisions and/or files.  OPTS controls the format of the diff
   (it contains options such as "-w -c", &c), or "" for the default.
```

```
    OPTIONS controls keyword expansion, as a string starting with "-k",
    or "" to use the default.  REV1 is the first revision to compare
    against; it must be non-NULL.  If REV2 is non-NULL, compare REV1
    and REV2; if REV2 is NULL compare REV1 with the file in the working
    directory, whose name is WORKFILE.  LABEL1 and LABEL2 are default
    file labels, and (if non-NULL) should be added as -L options
    to diff.  Output goes to stdout.

    Return value is 0 for success, -1 for a failure which set errno,
    or positive for a failure which printed a message on stderr.

    This used to exec rcsdiff, but now calls RCS_checkout and diff_exec.

    An issue is what timezone is used for the dates which appear in the
    diff output.  rcsdiff uses the -z flag, which is not presently
    processed by CVS diff, but I'm not sure exactly how hard to worry
    about this--any such features are undocumented in the context of
    CVS, and I'm not sure how important to users.  */
int
RCS_exec_rcsdiff (rcsfile, opts, options, rev1, rev2, label1, label2, workfile)
    RCSNode *rcsfile;
    char *opts;
    char *options;
    char *rev1;
    char *rev2;
    char *label1;
    char *label2;
    char *workfile;
{
    char *tmpfile1;
    char *tmpfile2;
    char *use_file2;
    int status, retval;

    tmpfile1 = cvs_temp_name ();
    tmpfile2 = NULL;

    cvs_output ("\
===================================================================\n\
RCS file: ", 0);
    cvs_output (rcsfile->path, 0);
    cvs_output ("\n", 1);

    /* Historically, 'cvs diff' has expanded the $Name keyword to the
       empty string when checking out revisions.  This is an accident,
       but no one has considered the issue thoroughly enough to determine
       what the best behavior is.  Passing NULL for the 'nametag' argument
       preserves the existing behavior. */

    cvs_output ("retrieving revision ", 0);
    cvs_output (rev1, 0);
    cvs_output ("\n", 1);
    status = RCS_checkout (rcsfile, NULL, rev1, NULL, options, tmpfile1,
                           (RCSCHECKOUTPROC)0, NULL);
    if (status > 0)
    {
        retval = status;
        goto error_return;
    }
    else if (status < 0)
    {
        error (0, errno,
               "cannot check out revision %s of %s", rev1, rcsfile->path);
        retval = 1;
        goto error_return;
    }

    if (rev2 == NULL)
    {
        assert (workfile != NULL);
        use_file2 = workfile;
    }
    else
    {
        tmpfile2 = cvs_temp_name ();
        cvs_output ("retrieving revision ", 0);
        cvs_output (rev2, 0);
        cvs_output ("\n", 1);
        status = RCS_checkout (rcsfile, NULL, rev2, NULL, options,
                               tmpfile2, (RCSCHECKOUTPROC)0, NULL);
        if (status > 0)
        {
            retval = status;
            goto error_return;
        }
        else if (status < 0)
        {
            error (0, errno,
                   "cannot check out revision %s of %s", rev2, rcsfile->path);
            return 1;
```

```
        }
450     use_file2 = tmpfile2;
    }

    RCS_output_diff_options (opts, rev1, rev2, workfile);
    status = diff_execv (tmpfile1, use_file2, label1, label2, opts, RUN_TTY);
    if (status >= 0)
    {
        retval = status;
        goto error_return;
    }
460 else if (status < 0)
    {
        error (0, errno,
               "cannot diff %s and %s", tmpfile1, use_file2);
        retval = 1;
        goto error_return;
    }

 error_return:
    {
470     int save_noexec = noexec;
        noexec = 0;
        if (unlink_file (tmpfile1) < 0)
        {
            if (!existence_error (errno))
                error (0, errno, "cannot remove temp file %s", tmpfile1);
        }
        noexec = save_noexec;
    }
    free (tmpfile1);
480 if (tmpfile2 != NULL)
    {
        int save_noexec = noexec;
        noexec = 0;
        if (unlink_file (tmpfile2) < 0)
        {
            if (!existence_error (errno))
                error (0, errno, "cannot remove temp file %s", tmpfile2);
        }
        noexec = save_noexec;
490     free (tmpfile2);
    }

    return retval;
}


/* Show differences between two files.  This is the start of a diff library.

   Some issues:
500

   * Should option parsing be part of the library or the caller?  The
   former allows the library to add options without changing the callers,
   but it causes various problems.  One is that something like --brief really
   wants special handling in CVS, and probably the caller should retain
   some flexibility in this area.  Another is online help (the library could
   have some feature for providing help, but how does that interact with
   the help provided by the caller directly?).  Another is that as things
   stand currently, there is no separate namespace for diff options versus
   "cvs diff" options like -l (that is, if the library adds an option which
510 conflicts with a CVS option, it is trouble).

   * This isn't required for a first-cut diff library, but if there
   would be a way for the caller to specify the timestamps that appear
   in the diffs (rather than the library getting them from the files),
   that would clean up the kludgy utime() calls in patch.c.

   Show differences between FILE1 and FILE2.  Either one can be
   DEVNULL to indicate a nonexistent file (same as an empty file
   currently, I suspect, but that may be an issue in and of itself).
520 OPTIONS is a list of diff options, or "" if none.  At a minimum,
   CVS expects that -c (update.c, patch.c) and -n (update.c) will be
   supported.  Other options, like -u, --speed-large-files, &c, will
   be specified if the user specified them.

   OUT is a filename to send the diffs to, or RUN_TTY to send them to
   stdout.  Error messages go to stderr.  Return value is 0 for
   success, -1 for a failure which set errno, 1 for success (and some
   differences were found), or >1 for a failure which printed a
   message on stderr.  */
530
int
diff_exec (file1, file2, options, out)
    char *file1;
    char *file2;
    char *options;
    char *out;
{
    char *args;
```

```
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* If either file1 or file2 are special files, pretend they are
       /dev/null.  Reason: suppose a file that represents a block
       special device in one revision becomes a regular file.  CVS
       must find the 'difference' between these files, but a special
       file contains no data useful for calculating this metric.  The
       safe thing to do is to treat the special file as an empty file,
       thus recording the regular file's full contents.  Doing so will
       create extremely large deltas at the point of transition
       between device files and regular files, but this is probably
       very rare anyway.

       There may be ways around this, but I think they are fraught
       with danger. -twp */

    if (preserve_perms &&
        strcmp (file1, DEVNULL) != 0 &&
        strcmp (file2, DEVNULL) != 0)
    {
        struct stat sb1, sb2;

        if (CVS_LSTAT (file1, &sb1) < 0)
            error (1, errno, "cannot get file information for %s", file1);
        if (CVS_LSTAT (file2, &sb2) < 0)
            error (1, errno, "cannot get file information for %s", file2);

        if (!S_ISREG (sb1.st_mode) && !S_ISDIR (sb1.st_mode))
            file1 = DEVNULL;
        if (!S_ISREG (sb2.st_mode) && !S_ISDIR (sb2.st_mode))
            file2 = DEVNULL;
    }
#endif

    args = xmalloc (strlen (options) + 10);
    /* The first word in this string is used only for error reporting. */
    sprintf (args, "diff %s", options);
    call_diff_setup (args);
    call_diff_arg (file1);
    call_diff_arg (file2);
    free (args);

    return call_diff (out);
}

int
diff_execv (file1, file2, label1, label2, options, out)
    char *file1;
    char *file2;
    char *label1;
    char *label2;
    char *options;
    char *out;
{
    char *args;

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    /* Pretend that special files are /dev/null for purposes of making
       diffs.  See comments in diff_exec. */

    if (preserve_perms &&
        strcmp (file1, DEVNULL) != 0 &&
        strcmp (file2, DEVNULL) != 0)
    {
        struct stat sb1, sb2;

        if (CVS_LSTAT (file1, &sb1) < 0)
            error (1, errno, "cannot get file information for %s", file1);
        if (CVS_LSTAT (file2, &sb2) < 0)
            error (1, errno, "cannot get file information for %s", file2);

        if (!S_ISREG (sb1.st_mode) && !S_ISDIR (sb1.st_mode))
            file1 = DEVNULL;
        if (!S_ISREG (sb2.st_mode) && !S_ISDIR (sb2.st_mode))
            file2 = DEVNULL;
    }
#endif

    args = xmalloc (strlen (options) + 10);
    /* The first word in this string is used only for error reporting.  */
    /* I guess we are pretty confident that options starts with a space.  */
    sprintf (args, "diff%s", options);
    call_diff_setup (args);
    if (label1)
        call_diff_arg (label1);
    if (label2)
        call_diff_arg (label2);
    call_diff_arg (file1);
    call_diff_arg (file2);
    free (args);
```

```
630     return call_diff (out);
}

/* Print the options passed to DIFF, in the format used by rcsdiff.
   The rcsdiff code that produces this output is extremely hairy, and
   it is not clear how rcsdiff decides which options to print and
   which not to print.  The code below reproduces every rcsdiff run
   that I have seen. */

static void
640 RCS_output_diff_options (opts, rev1, rev2, workfile)
    char *opts;
    char *rev1;
    char *rev2;
    char *workfile;
{
    char *tmp;

    tmp = (char *) xmalloc (strlen (opts) + strlen (rev1) + 10);

650 sprintf (tmp, "diff%s -r%s", opts, rev1);
    cvs_output (tmp, 0);
    free (tmp);

    if (rev2)
    {
        cvs_output (" -r", 3);
        cvs_output (rev2, 0);
    }
    else
660 {
        assert (workfile != NULL);
        cvs_output (" ", 1);
        cvs_output (workfile, 0);
    }
    cvs_output ("\n", 1);
}
```

A.47 recurse.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * General recursion handler
 *
 */

#include "cvs.h"
#include "savecwd.h"
#include "fileattr.h"
#include "edit.h"

#ifdef CLIENT_SUPPORT
static int do_argument_proc PROTO((Node * p, void *closure));
#endif
static int do_dir_proc PROTO((Node * p, void *closure));
static int do_file_proc PROTO((Node * p, void *closure));
static void addlist PROTO((List ** listp, char *key));
static int unroll_files_proc PROTO((Node *p, void *closure));
static void addfile PROTO((List **listp, char *dir, char *file));

static char *update_dir;
static char *repository = NULL;
static List *filelist = NULL; /* holds list of files on which to operate */
static List *dirlist = NULL; /* holds list of directories on which to operate */

struct recursion_frame {
    FILEPROC fileproc;
    FILESDONEPROC filesdoneproc;
    DIRENTPROC direntproc;
    DIRLEAVEPROC dirleaveproc;
    void *callerdat;
    Dtype flags;
    int which;
    int aflag;
    int readlock;
    int dosrcs;
};

static int do_recursion PROTO ((struct recursion_frame *frame));

/* I am half tempted to shove a struct file_info * into the struct
   recursion_frame (but then we would need to modify or create a
   recursion_frame for each file), or shove a struct recursion_frame *
   into the struct file_info (more tempting, although it isn't completely
   clear that the struct file_info should contain info about recursion
   processor internals).  So instead use this struct.  */

struct frame_and_file {
    struct recursion_frame *frame;
    struct file_info *finfo;
};

/* Similarly, we need to pass the entries list to do_dir_proc.  */

struct frame_and_entries {
    struct recursion_frame *frame;
    List *entries;
};

#ifdef CLIENT_SUPPORT
/* This is a callback to send "Argument" commands to the server in the
   case we've done a "cvs update" or "cvs commit" in a top-level
   directory where there is no CVSADM directory. */

static int
do_argument_proc (p, closure)
    Node *p;
    void *closure;
{
    char *dir = p->key;
    send_to_server ("Argument ", 0);
    send_to_server (dir, 0);
    send_to_server ("\012", 1);
    return 0;
}
#endif

/* Start a recursive command.

   Command line arguments (ARGC, ARGV) dictate the directories and
   files on which we operate.  In the special case of no arguments, we
   default to ".".  */
int
start_recursion (fileproc, filesdoneproc, direntproc, dirleaveproc, callerdat,
```

```
                argc, argv, local, which, aflag, readlock,
                update_preload, dosrcs)
    FILEPROC fileproc;
    FILESDONEPROC filesdoneproc;
    DIRENTPROC  direntproc;
    DIRLEAVEPROC dirleaveproc;
    void *callerdat;

    int argc;
    char **argv;
    int local;

    /* This specifies the kind of recursion.  There are several cases:

       1.  W_LOCAL is not set but W_REPOS or W_ATTIC is.  The current
       directory when we are called must be the repository and
       recursion proceeds according to what exists in the repository.

       2a.  W_LOCAL is set but W_REPOS and W_ATTIC are not.  The
       current directory when we are called must be the working
       directory.  Recursion proceeds according to what exists in the
       working directory, never (I think) consulting any part of the
       repository which does not correspond to the working directory
       ("correspond" == Name_Repository).

       2b.  W_LOCAL is set and so is W_REPOS or W_ATTIC.  This is the
       weird one.  The current directory when we are called must be
       the working directory.  We recurse through working directories,
       but we recurse into a directory if it is exists in the working
       directory *or* it exists in the repository.  If a directory
       does not exist in the working directory, the direntproc must
       either tell us to skip it (R_SKIP_ALL), or must create it (I
       think those are the only two cases).  */
    int which;

    int aflag;
    int readlock;
    char *update_preload;
    int dosrcs;
{
    int i, err = 0;
    List *files_by_dir = NULL;
    struct recursion_frame frame;

    frame.fileproc = fileproc;
    frame.filesdoneproc = filesdoneproc;
    frame.direntproc = direntproc;
    frame.dirleaveproc = dirleaveproc;
    frame.callerdat = callerdat;
    frame.flags = local ? R_SKIP_DIRS : R_PROCESS;
    frame.which = which;
    frame.aflag = aflag;
    frame.readlock = readlock;
    frame.dosrcs = dosrcs;

    expand_wild (argc, argv, &argc, &argv);

    if (update_preload == NULL)
        update_dir = xstrdup ("");
    else
        update_dir = xstrdup (update_preload);

    /* clean up from any previous calls to start_recursion */
    if (repository)
    {
        free (repository);
        repository = (char *) NULL;
    }
    if (filelist)
        dellist (&filelist); /* FIXME-krp: no longer correct. */
    if (dirlist)
        dellist (&dirlist);

#ifdef SERVER_SUPPORT
    if (server_active)
    {
        for (i = 0; i < argc; ++i)
            server_pathname_check (argv[i]);
    }
#endif

    if (argc == 0)
    {

        /*
         * There were no arguments, so we'll probably just recurse. The
         * exception to the rule is when we are called from a directory
         * without any CVS administration files.  That has always meant to
         * process each of the sub-directories, so we pretend like we were
         * called with the list of sub-dirs of the current dir as args
```

```
         */
180      if ((which & W_LOCAL) && !isdir (CVSADM))
         {
            dirlist = Find_Directories ((char *) NULL, W_LOCAL, (List *) NULL);
            /* If there are no sub-directories, there is a certain logic in
               favor of doing nothing, but in fact probably the user is just
               confused about what directory they are in, or whether they
               cvs add'd a new directory.  In the case of at least one
               sub-directory, at least when we recurse into them we
               notice (hopefully) whether they are under CVS control.  */
            if (list_isempty (dirlist))
190         {
               if (update_dir[0] == '\0')
                  error (0, 0, "in directory .:");
               else
                  error (0, 0, "in directory %s:", update_dir);
               error (1, 0,
                      "there is no version here; run '%s checkout' first",
                      program_name);
            }
#ifdef CLIENT_SUPPORT
200         else if (client_active && server_started)
            {
               /* In the the case "cvs update foo bar baz", a call to
                  send_file_names in update.c will have sent the
                  appropriate "Argument" commands to the server.  In
                  this case, that won't have happened, so we need to
                  do it here.  While this example uses "update", this
                  generalizes to other commands. */

               err += walklist (dirlist, do_argument_proc, NULL);
210         }
#endif
         }
         else
            addlist (&dirlist, ".");

         err += do_recursion (&frame);
         goto out;
      }

220
      /*
       * There were arguments, so we have to handle them by hand. To do
       * that, we set up the filelist and dirlist with the arguments and
       * call do_recursion.  do_recursion recognizes the fact that the
       * lists are non-null when it starts and doesn't update them.
       *
       * explicitly named directories are stored in dirlist.
       * explicitly named files are stored in filelist.
       * other possibility is named entities whicha are not currently in
230    * the working directory.
       */

      for (i = 0; i < argc; i++)
      {
         /* if this argument is a directory, then add it to the list of
            directories. */

         if (!wrap_name_has (argv[i], WRAP_TOCVS) && isdir (argv[i]))
            addlist (&dirlist, argv[i]);
240      else
         {
            /* otherwise, split argument into directory and component names. */
            char *dir;
            char *comp;
            char *file_to_try;

            /* Now break out argv[i] into directory part (DIR) and file part (COMP).
                  DIR and COMP will each point to a newly malloc'd string.  */
            dir = xstrdup (argv[i]);
250         comp = last_component (dir);
            if (comp == dir)
            {
               /* no dir component.  What we have is an implied "./" */
               dir = xstrdup(".");
            }
            else
            {
               char *p = comp;

260            p[-1] = '\0';
               comp = xstrdup (p);
            }

            /* if this argument exists as a file in the current
               working directory tree, then add it to the files list.  */

            if (!(which & W_LOCAL))
            {
```

```
            /* If doing rtag, we've done a chdir to the repository. */
            file_to_try = xmalloc (strlen (argv[i]) + sizeof (RCSEXT) + 5);
            sprintf (file_to_try, "%s%s", argv[i], RCSEXT);
        }
        else
            file_to_try = xstrdup (argv[i]);

        if (isfile (file_to_try))
            addfile (&files_by_dir, dir, comp);
        else if (isdir (dir))
        {
            if ((which & W_LOCAL) && isdir (CVSADM)
#ifdef CLIENT_SUPPORT
                && !client_active
#endif
                )
            {
                /* otherwise, look for it in the repository. */
                char *tmp_update_dir;
                char *repos;
                char *reposfile;

                tmp_update_dir = xmalloc (strlen (update_dir)
                                          + strlen (dir)
                                          + 5);
                strcpy (tmp_update_dir, update_dir);

                if (*tmp_update_dir != '\0')
                    (void) strcat (tmp_update_dir, "/");

                (void) strcat (tmp_update_dir, dir);

                /* look for it in the repository. */
                repos = Name_Repository (dir, tmp_update_dir);
                reposfile = xmalloc (strlen (repos)
                                     + strlen (comp)
                                     + 5);
                (void) sprintf (reposfile, "%s/%s", repos, comp);
                free (repos);

                if (!wrap_name_has (comp, WRAP_TOCVS) && isdir (reposfile))
                    addlist (&dirlist, argv[i]);
                else
                    addfile (&files_by_dir, dir, comp);

                free (tmp_update_dir);
                free (reposfile);
            }
            else
                addfile (&files_by_dir, dir, comp);
        }
        else
            error (1, 0, "no such directory `%s'", dir);

        free (file_to_try);
        free (dir);
        free (comp);
      }
    }

    /* At this point we have looped over all named arguments and built
       a coupla lists.  Now we unroll the lists, setting up and
       calling do_recursion. */

    err += walklist (files_by_dir, unroll_files_proc, (void *) &frame);
    dellist(&files_by_dir);

    /* then do_recursion on the dirlist. */
    if (dirlist != NULL)
        err += do_recursion (&frame);

    /* Free the data which expand_wild allocated.  */
    free_names (&argc, argv);

 out:
    free (update_dir);
    update_dir = NULL;
    return (err);
}

/*
 * Implement the recursive policies on the local directory.  This may be
 * called directly, or may be called by start_recursion
 */
static int
do_recursion (frame)
    struct recursion_frame *frame;
{
    int err = 0;
    int dodoneproc = 1;
```

```
    char *srepository;
    List *entries = NULL;
    int should_readlock;

    /* do nothing if told */
    if (frame->flags == R_SKIP_ALL)
        return (0);

    should_readlock = noexec ? 0 : frame->readlock;

    /* The fact that locks are not active here is what makes us fail to have
       the

           If someone commits some changes in one cvs command,
           then an update by someone else will either get all the
           changes, or none of them.

       property (see node Concurrency in cvs.texinfo).

       The most straightforward fix would just to readlock the whole
       tree before starting an update, but that means that if a commit
       gets blocked on a big update, it might need to wait a *long*
       time.

       A more adequate fix would be a two-pass design for update,
       checkout, etc.  The first pass would go through the repository,
       with the whole tree readlocked, noting what versions of each
       file we want to get.  The second pass would release all locks
       (except perhaps short-term locks on one file at a
       time--although I think RCS already deals with this) and
       actually get the files, specifying the particular versions it wants.

       This could be sped up by separating out the data needed for the
       first pass into a separate file(s)--for example a file
       attribute for each file whose value contains the head revision
       for each branch.  The structure should be designed so that
       commit can relatively quickly update the information for a
       single file or a handful of files (file attributes, as
       implemented in Jan 96, are probably acceptable; improvements
       would be possible such as branch attributes which are in
       separate files for each branch).  */

#if defined(SERVER_SUPPORT) && defined(SERVER_FLOWCONTROL)
    /*
     * Now would be a good time to check to see if we need to stop
     * generating data, to give the buffers a chance to drain to the
     * remote client.  We should not have locks active at this point.
     */
    if (server_active
        /* If there are writelocks around, we cannot pause here.  */
        && (should_readlock || noexec))
        server_pause_check();
#endif

    /*
     * Fill in repository with the current repository
     */
    if (frame->which & W_LOCAL)
    {
        if (isdir (CVSADM))
            repository = Name_Repository ((char *) NULL, update_dir);
        else
            repository = NULL;
    }
    else
    {
        repository = xgetwd ();
        if (repository == NULL)
            error (1, errno, "could not get working directory");
    }
    srepository = repository;      /* remember what to free */

    fileattr_startdir (repository);

    /*
     * The filesdoneproc needs to be called for each directory where files
     * processed, or each directory that is processed by a call where no
     * directories were passed in.  In fact, the only time we don't want to
     * call back the filesdoneproc is when we are processing directories that
     * were passed in on the command line (or in the special case of '.' when
     * we were called with no args
     */
    if (dirlist != NULL && filelist == NULL)
        dodoneproc = 0;

    /*
     * If filelist or dirlist is already set, we don't look again. Otherwise,
     * find the files and directories
     */
    if (filelist == NULL && dirlist == NULL)
```

```
    {
450     /* both lists were NULL, so start from scratch */
        if (frame->fileproc != NULL && frame->flags != R_SKIP_FILES)
        {
            int lwhich = frame->which;

            /* be sure to look in the attic if we have sticky tags/date */
            if ((lwhich & W_ATTIC) == 0)
                if (isreadable (CVSADM_TAG))
                    lwhich |= W_ATTIC;

460         /* In the !(which & W_LOCAL) case, we filled in repository
               earlier in the function.  In the (which & W_LOCAL) case,
               the Find_Names function is going to look through the
               Entries file.  If we do not have a repository, that
               does not make sense, so we insist upon having a
               repository at this point.  Name_Repository will give a
               reasonable error message.  */
            if (repository == NULL)
                repository = Name_Repository ((char *) NULL, update_dir);

470         /* find the files and fill in entries if appropriate */
            filelist = Find_Names (repository, lwhich, frame->aflag, &entries);
        }

        /* find sub-directories if we will recurse */
        if (frame->flags != R_SKIP_DIRS)
            dirlist = Find_Directories (repository, frame->which, entries);
    }
    else
    {
480     /* something was passed on the command line */
        if (filelist != NULL && frame->fileproc != NULL)
        {
            /* we will process files, so pre-parse entries */
            if (frame->which & W_LOCAL)
                entries = Entries_Open (frame->aflag, NULL);
        }
    }

    /* process the files (if any) */
490 if (filelist != NULL && frame->fileproc)
    {
        struct file_info finfo_struct;
        struct frame_and_file frfile;

        /* read lock it if necessary */
        if (should_readlock && repository && Reader_Lock (repository) != 0)
            error (1, 0, "read lock failed - giving up");

#ifdef CLIENT_SUPPORT
500     /* For the server, we handle notifications in a completely different
           place (server_notify).  For local, we can't do them here--we don't
           have writelocks in place, and there is no way to get writelocks
           here.  */
        if (client_active)
            notify_check (repository, update_dir);
#endif /* CLIENT_SUPPORT */

        finfo_struct.repository = repository;
        finfo_struct.update_dir = update_dir;
510     finfo_struct.entries = entries;
        /* do_file_proc will fill in finfo_struct.file.  */

        frfile.finfo = &finfo_struct;
        frfile.frame = frame;

        /* process the files */
        err += walklist (filelist, do_file_proc, &frfile);

        /* unlock it */
520     if (should_readlock)
            Lock_Cleanup ();

        /* clean up */
        dellist (&filelist);
    }

    /* call-back files done proc (if any) */
    if (dodoneproc && frame->filesdoneproc != NULL)
        err = frame->filesdoneproc (frame->callerdat, err, repository,
530                                 update_dir[0] ? update_dir : ".",
                                    entries);

    fileattr_write ();
    fileattr_free ();

    /* process the directories (if necessary) */
    if (dirlist != NULL)
    {
```

```
        struct frame_and_entries frent;

        frent.frame = frame;
        frent.entries = entries;
        err += walklist (dirlist, do_dir_proc, (void *) &frent);
    }
#if 0
    else if (frame->dirleaveproc != NULL)
        err += frame->dirleaveproc (frame->callerdat, ".", err, ".");
#endif
    dellist (&dirlist);

    if (entries)
    {
        Entries_Close (entries);
        entries = NULL;
    }

    /* free the saved copy of the pointer if necessary */
    if (srepository)
    {
        free (srepository);
        repository = (char *) NULL;
    }

    return (err);
}

/*
 * Process each of the files in the list with the callback proc
 */
static int
do_file_proc (p, closure)
    Node *p;
    void *closure;
{
    struct frame_and_file *frfile = (struct frame_and_file *)closure;
    struct file_info *finfo = frfile->finfo;
    int ret;

    finfo->file = p->key;
    finfo->fullname = xmalloc (strlen (finfo->file)
                               + strlen (finfo->update_dir)
                               + 2);
    finfo->fullname[0] = '\0';
    if (finfo->update_dir[0] != '\0')
    {
        strcat (finfo->fullname, finfo->update_dir);
        strcat (finfo->fullname, "/");
    }
    strcat (finfo->fullname, finfo->file);

    if (frfile->frame->dosrcs && repository)
        finfo->rcs = RCS_parse (finfo->file, repository);
    else
        finfo->rcs = (RCSNode *) NULL;
    ret = frfile->frame->fileproc (frfile->frame->callerdat, finfo);

    freercsnode(&finfo->rcs);
    free (finfo->fullname);

    /* Allow the user to monitor progress with tail -f.  Doing this once
       per file should be no big deal, but we don't want the performance
       hit of flushing on every line like previous versions of CVS.  */
    cvs_flushout ();

    return (ret);
}

/*
 * Process each of the directories in the list (recursing as we go)
 */
static int
do_dir_proc (p, closure)
    Node *p;
    void *closure;
{
    struct frame_and_entries *frent = (struct frame_and_entries *) closure;
    struct recursion_frame *frame = frent->frame;
    struct recursion_frame xframe;
    char *dir = p->key;
    char *newrepos;
    List *sdirlist;
    char *srepository;
    Dtype dir_return = R_PROCESS;
    int stripped_dot = 0;
    int err = 0;
    struct saved_cwd cwd;
    char *saved_update_dir;
```

```
    if (fncmp (dir, CVSADM) == 0)
630 {
        /* This seems to most often happen when users (beginning users,
           generally), try "cvs ci *" or something similar.  On that
           theory, it is possible that we should just silently skip the
           CVSADM directories, but on the other hand, using a wildcard
           like this isn't necessarily a practice to encourage (it operates
           only on files which exist in the working directory, unlike
           regular CVS recursion).  */

        /* FIXME-reentrancy: printed_cvs_msg should be in a "command
640        struct" or some such, so that it gets cleared for each new
           command (this is possible using the remote protocol and a
           custom-written client).  The struct recursion_frame is not
           far back enough though, some commands (commit at least)
           will call start_recursion several times.  An alternate solution
           would be to take this whole check and move it to a new function
           validate_arguments or some such that all the commands call
           and which snips the offending directory from the argc,argv
           vector.  */
        static int printed_cvs_msg = 0;
650     if (!printed_cvs_msg)
        {
            error (0, 0, "warning: directory %s specified in argument",
                   dir);
            error (0, 0, "\
but CVS uses %s for its own purposes; skipping %s directory",
                   CVSADM, dir);
            printed_cvs_msg = 1;
        }
        return 0;
660 }

    saved_update_dir = update_dir;
    update_dir = xmalloc (strlen (saved_update_dir)
                          + strlen (dir)
                          + 5);
    strcpy (update_dir, saved_update_dir);

    /* set up update_dir - skip dots if not at start */
    if (strcmp (dir, ".") != 0)
670 {
        if (update_dir[0] != '\0')
        {
            (void) strcat (update_dir, "/");
            (void) strcat (update_dir, dir);
        }
        else
            (void) strcpy (update_dir, dir);

        /*
680      * Here we need a plausible repository name for the sub-directory. We
         * create one by concatenating the new directory name onto the
         * previous repository name.  The only case where the name should be
         * used is in the case where we are creating a new sub-directory for
         * update -d and in that case the generated name will be correct.
         */
        if (repository == NULL)
            newrepos = xstrdup ("");
        else
        {
690         newrepos = xmalloc (strlen (repository) + strlen (dir) + 5);
            sprintf (newrepos, "%s/%s", repository, dir);
        }
    }
    else
    {
        if (update_dir[0] == '\0')
            (void) strcpy (update_dir, dir);

        if (repository == NULL)
700         newrepos = xstrdup ("");
        else
            newrepos = xstrdup (repository);
    }

    /* Check to see that the CVSADM directory, if it exists, seems to be
       well-formed.  It can be missing files if the user hit ^C in the
       middle of a previous run.  We want to (a) make this a nonfatal
       error, and (b) make sure we print which directory has the
       problem.
710
       Do this before the direntproc, so that (1) the direntproc
       doesn't have to guess/deduce whether we will skip the directory
       (e.g. send_dirent_proc and whether to send the directory), and
       (2) so that the warm fuzzy doesn't get printed if we skip the
       directory.  */
    if (frame->which & W_LOCAL)
    {
        char *cvsadmdir;
```

```
720         cvsadmdir = xmalloc (strlen (dir)
                                 + sizeof (CVSADM_REP)
                                 + sizeof (CVSADM_ENT)
                                 + 80);

            strcpy (cvsadmdir, dir);
            strcat (cvsadmdir, "/");
            strcat (cvsadmdir, CVSADM);
            if (isdir (cvsadmdir))
            {
730             strcpy (cvsadmdir, dir);
                strcat (cvsadmdir, "/");
                strcat (cvsadmdir, CVSADM_REP);
                if (!isfile (cvsadmdir))
                {
                    /* Some commands like update may have printed "? foo" but
                       if we were planning to recurse, and don't on account of
                       CVS/Repository, we want to say why.  */
                    error (0, 0, "ignoring %s (%s missing)", update_dir,
                           CVSADM_REP);
740                 dir_return = R_SKIP_ALL;
                }

                /* Likewise for CVS/Entries.  */
                if (dir_return != R_SKIP_ALL)
                {
                    strcpy (cvsadmdir, dir);
                    strcat (cvsadmdir, "/");
                    strcat (cvsadmdir, CVSADM_ENT);
                    if (!isfile (cvsadmdir))
750                 {
                        /* Some commands like update may have printed "? foo" but
                           if we were planning to recurse, and don't on account of
                           CVS/Repository, we want to say why.  */
                        error (0, 0, "ignoring %s (%s missing)", update_dir,
                               CVSADM_ENT);
                        dir_return = R_SKIP_ALL;
                    }
                }
            }
760         free (cvsadmdir);
        }

        /* call-back dir entry proc (if any) */
        if (dir_return == R_SKIP_ALL)
            ;
        else if (frame->direntproc != NULL)
            dir_return = frame->direntproc (frame->callerdat, dir, newrepos,
                                            update_dir, frent->entries);
        else
770     {
            /* Generic behavior.  I don't see a reason to make the caller specify
               a direntproc just to get this.  */
            if ((frame->which & W_LOCAL) && !isdir (dir))
                dir_return = R_SKIP_ALL;
        }

        free (newrepos);

        /* only process the dir if the return code was 0 */
780     if (dir_return != R_SKIP_ALL)
        {
            /* save our current directory and static vars */
            if (save_cwd (&cwd))
                error_exit ();
            sdirlist = dirlist;
            srepository = repository;
            dirlist = NULL;

            /* cd to the sub-directory */
790         if ( CVS_CHDIR (dir) < 0)
                error (1, errno, "could not chdir to %s", dir);

            /* honor the global SKIP_DIRS (a.k.a. local) */
            if (frame->flags == R_SKIP_DIRS)
                dir_return = R_SKIP_DIRS;

            /* remember if the '.' will be stripped for subsequent dirs */
            if (strcmp (update_dir, ".") == 0)
            {
800             update_dir[0] = '\0';
                stripped_dot = 1;
            }

            /* make the recursive call */
            xframe = *frame;
            xframe.flags = dir_return;
            err += do_recursion (&xframe);
```

```
        /* put the '.' back if necessary */
        if (stripped_dot)
            (void) strcpy (update_dir, ".");

        /* call-back dir leave proc (if any) */
        if (frame->dirleaveproc != NULL)
            err = frame->dirleaveproc (frame->callerdat, dir, err, update_dir,
                                       frent->entries);

        /* get back to where we started and restore state vars */
        if (restore_cwd (&cwd, NULL))
            error_exit ();
        free_cwd (&cwd);
        dirlist = sdirlist;
        repository = srepository;
    }

    free (update_dir);
    update_dir = saved_update_dir;

    return (err);
}

/*
 * Add a node to a list allocating the list if necessary.
 */
static void
addlist (listp, key)
    List **listp;
    char *key;
{
    Node *p;

    if (*listp == NULL)
        *listp = getlist ();
    p = getnode ();
    p->type = FILES;
    p->key = xstrdup (key);
    if (addnode (*listp, p) != 0)
        freenode (p);
}

static void
addfile (listp, dir, file)
    List **listp;
    char *dir;
    char *file;
{
    Node *n;

    /* add this dir. */
    addlist (listp, dir);

    n = findnode (*listp, dir);
    if (n == NULL)
    {
        error (1, 0, "can't find recently added dir node `%s' in start_recursion.",
               dir);
    }

    n->type = DIRS;
    addlist ((List **) &n->data, file);
    return;
}

static int
unroll_files_proc (p, closure)
    Node *p;
    void *closure;
{
    Node *n;
    struct recursion_frame *frame = (struct recursion_frame *) closure;
    int err = 0;
    List *save_dirlist;
    char *save_update_dir = NULL;
    struct saved_cwd cwd;

    /* if this dir was also an explicitly named argument, then skip
       it.  We'll catch it later when we do dirs. */
    n = findnode (dirlist, p->key);
    if (n != NULL)
        return (0);

    /* otherwise, call dorecusion for this list of files. */
    filelist = (List *) p->data;
    p->data = NULL;
    save_dirlist = dirlist;
    dirlist = NULL;

    if (strcmp(p->key, ".") != 0)
```

```
    {
900     if (save_cwd (&cwd))
            error_exit ();
        if ( CVS_CHDIR (p->key) < 0)
            error (1, errno, "could not chdir to %s", p->key);

        save_update_dir = update_dir;
        update_dir = xmalloc (strlen (save_update_dir)
                              + strlen (p->key)
                              + 5);
        strcpy (update_dir, save_update_dir);
910
        if (*update_dir != '\0')
            (void) strcat (update_dir, "/");

        (void) strcat (update_dir, p->key);
    }

    err += do_recursion (frame);

    if (save_update_dir != NULL)
920 {
        free (update_dir);
        update_dir = save_update_dir;

        if (restore_cwd (&cwd, NULL))
            error_exit ();
        free_cwd (&cwd);
    }

    dirlist = save_dirlist;
930 filelist = NULL;
    return(err);
}
```

A.48 release.c

```
/*
 * Release: "cancel" a checkout in the history log.
 *
 * - Enter a line in the history log indicating the "release". - If asked to,
 * delete the local working directory.
 */

#include "cvs.h"
#include "getline.h"

static void release_delete PROTO((char *dir));

static const char *const release_usage[] =
{
    "Usage: %s %s [-d] directories. . .\n",
    "\t-d\tDelete the given directory.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

#ifdef SERVER_SUPPORT
static int release_server PROTO ((int argc, char **argv));

/* This is the server side of cvs release.  */
static int
release_server (argc, argv)
    int argc;
    char **argv;
{
    int i;

    /* Note that we skip argv[0].  */
    for (i = 1; i < argc; ++i)
        history_write ('F', argv[i], "", argv[i], "");
    return 0;
}

#endif /* SERVER_SUPPORT */

/* There are various things to improve about this implementation:

   1.  Using run_popen to run "cvs update" could be replaced by a
   fairly simple start_recursion/classify_file loop--a win for
   portability, performance, and cleanliness.  In particular, there is
   no particularly good way to find the right "cvs".

   2.  The fact that "cvs update" contacts the server slows things down;
   it undermines the case for using "cvs release" rather than "rm -rf".
   However, for correctly printing "? foo" and correctly handling
   CVSROOTADM_IGNORE, we currently need to contact the server.  (One
   idea for how to fix this is to stash a copy of CVSROOTADM_IGNORE in
   the working directories; see comment at base_* in entries.c for a
   few thoughts on that).

   3.  Would be nice to take processing things on the client side one step
   further, and making it like edit/unedit in terms of working well if
   disconnected from the network, and then sending a delayed
   notification.

   4.  Having separate network turnarounds for the "Notify" request
   which we do as part of unedit, and for the "release" itself, is slow
   and unnecessary.  */

int
release (argc, argv)
    int argc;
    char **argv;
{
    FILE *fp;
    int i, c;
    char *repository;
    char *line = NULL;
    size_t line_allocated = 0;
    char *update_cmd;
    char *thisarg;
    int arg_start_idx;
    int err = 0;
    short delete_flag = 0;

#ifdef SERVER_SUPPORT
    if (server_active)
        return release_server (argc, argv);
#endif

    /* Everything from here on is client or local.  */
    if (argc == -1)
        usage (release_usage);
    optind = 0;
```

```
    while ((c = getopt (argc, argv, "+Qdq")) != -1)
    {
        switch (c)
        {
            case 'Q':
            case 'q':
                error (1, 0,
                       "-q or -Q must be specified before \"%s\"",
                       command_name);
                break;
            case 'd':
                delete_flag++;
                break;
            case '?':
            default:
                usage (release_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    /* We're going to run "cvs -n -q update" and check its output; if
     * the output is sufficiently unalarming, then we release with no
     * questions asked.  Else we prompt, then maybe release.
     */
    /* Construct the update command. */
    update_cmd = xmalloc (strlen (program_path)
                          + strlen (CVSroot_original)
                          + 20);
    sprintf (update_cmd, "%s -n -q -d %s update",
             program_path, CVSroot_original);

#ifdef CLIENT_SUPPORT
    /* Start the server; we'll close it after looping. */
    if (client_active)
    {
        start_server ();
        ign_setup ();
    }
#endif /* CLIENT_SUPPORT */

    arg_start_idx = 0;

    for (i = arg_start_idx; i < argc; i++)
    {
        thisarg = argv[i];

        if (isdir (thisarg))
        {
            if (CVS_CHDIR (thisarg) < 0)
            {
                if (!really_quiet)
                    error (0, errno, "can't chdir to: %s", thisarg);
                continue;
            }
            if (!isdir (CVSADM))
            {
                if (!really_quiet)
                    error (0, 0, "no repository directory: %s", thisarg);
                continue;
            }
        }
        else
        {
            if (!really_quiet)
                error (0, 0, "no such directory: %s", thisarg);
            continue;
        }

        repository = Name_Repository ((char *) NULL, (char *) NULL);

        if (!really_quiet)
        {
            int line_length;

            /* The "release" command piggybacks on "update", which
               does the real work of finding out if anything is not
               up-to-date with the repository.  Then "release" prompts
               the user, telling her how many files have been
               modified, and asking if she still wants to do the
               release. */
            fp = run_popen (update_cmd, "r");
            if (fp == NULL)
                error (1, 0, "cannot run command %s", update_cmd);

            c = 0;

            while ((line_length = getline (&line, &line_allocated, fp)) >= 0)
            {
```

```
            if (strchr ("MARCZ", *line))
180             c++;
            (void) printf (line);
        }
        if (line_length < 0 && !feof (fp))
            error (0, errno, "cannot read from subprocess");

        /* If the update exited with an error, then we just want to
           complain and go on to the next arg.  Especially, we do
           not want to delete the local copy, since it's obviously
           not what the user thinks it is.  */
190     if ((pclose (fp)) != 0)
        {
            error (0, 0, "unable to release `%s'", thisarg);
            continue;
        }

        printf ("You have [%d] altered files in this repository.\n",
                c);
        printf ("Are you sure you want to release %sdirectory `%s': ",
                delete_flag ? "(and delete) " : "", thisarg);
200     c = !yesno ();
        if (c)                  /* "No" */
        {
            (void) fprintf (stderr, "** `%s' aborted by user choice.\n",
                            command_name);
            free (repository);
            continue;
        }
    }

210 if (1
#ifdef CLIENT_SUPPORT
        && !(client_active
             && (!supported_request ("noop")
                 || !supported_request ("Notify")))
#endif
        )
    {
        /* We are chdir'ed into the directory in question.
           So don't pass args to unedit.  */
220     int argc = 1;
        char *argv[3];
        argv[0] = "dummy";
        argv[1] = NULL;
        err += unedit (argc, argv);
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
230     send_to_server ("Argument ", 0);
        send_to_server (thisarg, 0);
        send_to_server ("\012", 1);
        send_to_server ("release\012", 0);
    }
    else
#endif /* CLIENT_SUPPORT */
    {
        history_write ('F', thisarg, "", thisarg, ""); /* F == Free */
    }
240
    free (repository);
    if (delete_flag) release_delete (thisarg);

#ifdef CLIENT_SUPPORT
    if (client_active)
        err += get_server_responses ();
#endif /* CLIENT_SUPPORT */
}

250 #ifdef CLIENT_SUPPORT
if (client_active)
{
    /* Unfortunately, client.c doesn't offer a way to close
       the connection without waiting for responses.  The extra
       network turnaround here is quite unnecessary other than
       that....  */
    send_to_server ("noop\012", 0);
    err += get_responses_and_close ();
}
260 #endif /* CLIENT_SUPPORT */

free (update_cmd);
if (line != NULL)
    free (line);
return err;
}
```

```
    /* We want to "rm -r" the working directory, but let us be a little
270    paranoid.  */
    static void
    release_delete (dir)
        char *dir;
    {
        struct stat st;
        ino_t ino;

        (void) CVS_STAT (".", &st);
        ino = st.st_ino;
280     (void) CVS_CHDIR (". .");
        (void) CVS_STAT (dir, &st);
        if (ino != st.st_ino)
        {
            /* This test does not work on cygwin32, because under cygwin32
               the st_ino field is not the same when you refer to a file
               by a different name.  This is a cygwin32 bug, but then I
               don't see what the point of this test is anyhow.  */
    #ifndef __CYGWIN32__
            error (0, 0,
290                 "Parent dir on a different disk, delete of %s aborted", dir);
            return;
    #endif
        }
        /*
         * XXX - shouldn't this just delete the CVS-controlled files and, perhaps,
         * the files that would normally be ignored and leave everything else?
         */
        if (unlink_file_dir (dir) < 0)
            error (0, errno, "deletion of directory %s failed", dir);
300 }
```

A.49 remove.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Remove a File
 *
 * Removes entries from the present version. The entries will be removed from
 * the RCS repository upon the next "commit".
 *
 * "remove" accepts no options, only file names that are to be removed.  The
 * file must not exist in the current directory for "remove" to work
 * correctly.
 */

#include "cvs.h"

#ifdef CLIENT_SUPPORT
static int remove_force_fileproc PROTO ((void *callerdat,
                                         struct file_info *finfo));
#endif
static int remove_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static Dtype remove_dirproc PROTO ((void *callerdat, char *dir,
                                    char *repos, char *update_dir,
                                    List *entries));

static int force;
static int local;
static int removed_files;
static int existing_files;

static const char *const remove_usage[] =
{
    "Usage: %s %s [-flR] [files...]\n",
    "\t-f\tDelete the file before removing it.\n",
    "\t-l\tProcess this directory only (not recursive).\n",
    "\t-R\tProcess directories recursively.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
cvsremove (argc, argv)
    int argc;
    char **argv;
{
    int c, err;

    if (argc == -1)
        usage (remove_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+flR")) != -1)
    {
        switch (c)
        {
            case 'f':
                force = 1;
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (remove_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    wrap_setup ();

#ifdef CLIENT_SUPPORT
    if (client_active) {
        /* Call expand_wild so that the local removal of files will
           work.  It's ok to do it always because we have to send the
           file names expanded anyway.  */
        expand_wild (argc, argv, &argc, &argv);

        if (force)
        {
            if (!noexec)
```

```
            {
                start_recursion (remove_force_fileproc, (FILESDONEPROC) NULL,
                                 (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL,
                                 (void *) NULL, argc, argv, local, W_LOCAL,
                                 0, 0, (char *) NULL, 0);
            }
            /* else FIXME should probably act as if the file doesn't exist
               in doing the following checks.  */
        }

        start_server ();
        ign_setup ();
        if (local)
            send_arg("-l");
        send_file_names (argc, argv, 0);
        /* FIXME: Can't we set SEND_NO_CONTENTS here?  Needs investigation.  */
        send_files (argc, argv, local, 0, 0);
        send_to_server ("remove\012", 0);
        return get_responses_and_close ();
    }
#endif

    /* start the recursion processor */
    err = start_recursion (remove_fileproc, (FILESDONEPROC) NULL,
                           remove_dirproc, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv,
                           local, W_LOCAL, 0, 1, (char *) NULL, 1);

    if (removed_files)
        error (0, 0, "use '%s commit' to remove %s permanently", program_name,
               (removed_files == 1) ? "this file" : "these files");

    if (existing_files)
        error (0, 0,
               ((existing_files == 1) ?
                "%d file exists; remove it first" :
                "%d files exist; remove them first"),
               existing_files);

    return (err);
}

#ifdef CLIENT_SUPPORT

/*
 * This is called via start_recursion if we are running as the client
 * and the -f option was used.  We just physically remove the file.
 */

/*ARGSUSED*/
static int
remove_force_fileproc (callerdat, finfo)
     void *callerdat;
     struct file_info *finfo;
{
    if (CVS_UNLINK (finfo->file) < 0 && ! existence_error (errno))
        error (0, errno, "unable to remove %s", finfo->fullname);
    return 0;
}

#endif

/*
 * remove the file, only if it has already been physically removed
 */
/* ARGSUSED */
static int
remove_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    Vers_TS *vers;

    if (force)
    {
        if (!noexec)
        {
            if ( CVS_UNLINK (finfo->file) < 0 && ! existence_error (errno))
            {
                error (0, errno, "unable to remove %s", finfo->fullname);
            }
        }
        /* else FIXME should probably act as if the file doesn't exist
           in doing the following checks.  */
    }

    vers = Version_TS (finfo, NULL, NULL, NULL, 0, 0);

    if (vers->ts_user != NULL)
    {
```

```
            existing_files++;
            if (!quiet)
                error (0, 0, "file `%s' still in working directory",
                       finfo->fullname);
        }
        else if (vers->vn_user == NULL)
        {
            if (!quiet)
                error (0, 0, "nothing known about `%s'", finfo->fullname);
        }
        else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '\0')
        {
            char *fname;

            /*
             * It's a file that has been added, but not commited yet. So,
             * remove the ,t file for it and scratch it from the
             * entries file. */
            Scratch_Entry (finfo->entries, finfo->file);
            fname = xmalloc (strlen (finfo->file)
                             + sizeof (CVSADM)
                             + sizeof (CVSEXT_LOG)
                             + 10);
            (void) sprintf (fname, "%s/%s%s", CVSADM, finfo->file, CVSEXT_LOG);
            (void) unlink_file (fname);
            if (!quiet)
                error (0, 0, "removed `%s'", finfo->fullname);

        #ifdef SERVER_SUPPORT
            if (server_active)
                server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
        #endif
            free (fname);
        }
        else if (vers->vn_user[0] == '-')
        {
            if (!quiet)
                error (0, 0, "file `%s' already scheduled for removal",
                       finfo->fullname);
        }
        else if (vers->tag != NULL && isdigit (*vers->tag))
        {
            /* Commit will just give an error, and so there seems to be
               little reason to allow the remove.  I mean, conflicts that
               arise out of parallel development are one thing, but conflicts
               that arise from sticky tags are quite another.

               I would have thought that non-branch sticky tags should be the
               same but at least now, removing a file with a non-branch sticky
               tag means to delete the tag from the file.  I'm not sure that
               is a good behavior, but until it is changed, we need to allow
               it.  */
            error (0, 0, "\
        cannot remove file `%s' which has a numeric sticky tag of `%s'",
                   finfo->fullname, vers->tag);
        }
        else
        {
            char *fname;

            /* Re-register it with a negative version number.  */
            fname = xmalloc (strlen (vers->vn_user) + 5);
            (void) strcpy (fname, "-");
            (void) strcat (fname, vers->vn_user);
            Register (finfo->entries, finfo->file, fname, vers->ts_rcs, vers->options,
                      vers->tag, vers->date, vers->ts_conflict, CVSroot_directory, finfo->repository);
            if (!quiet)
                error (0, 0, "scheduling `%s' for removal", finfo->fullname);
            removed_files++;

        #ifdef SERVER_SUPPORT
            if (server_active)
                server_checked_in (finfo->file, finfo->update_dir, finfo->repository);
        #endif
            free (fname);
        }

        freevers_ts (&vers);
        return (0);
    }

    /*
     * Print a warm fuzzy message
     */
    /* ARGSUSED */
    static Dtype
    remove_dirproc (callerdat, dir, repos, update_dir, entries)
        void *callerdat;
        char *dir;
        char *repos;
```

```
    char *update_dir;
270 List *entries;
{
    if (!quiet)
        error (0, 0, "Removing %s", update_dir);
    return (R_PROCESS);
}
```

A.50 repos.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include <assert.h>
#include "cvs.h"
#include "getline.h"

/* Determine the name of the RCS repository for directory DIR in the
   current working directory, or for the current working directory
   itself if DIR is NULL.  Returns the name in a newly-malloc'd
   string.  On error, gives a fatal error and does not return.
   UPDATE_DIR is the path from where cvs was invoked (for use in error
   messages), and should contain DIR as its last component.
   UPDATE_DIR can be NULL to signify the directory in which cvs was
   invoked.  */

char *
Name_Repository (dir, update_dir)
    char *dir;
    char *update_dir;
{
    FILE *fpin;
    char *xupdate_dir;
    char *repos = NULL;
    size_t repos_allocated = 0;
    char *tmp;
    char* tmp2;
    char *cp;

    if (update_dir && *update_dir)
        xupdate_dir = update_dir;
    else
        xupdate_dir = ".";

    if (dir != NULL)
    {
        tmp = xmalloc (strlen (dir) + sizeof (CVSADM_REP) + 10);
        (void) sprintf (tmp, "%s/%s", dir, CVSADM_REP);
    }
    else
        tmp = xstrdup (CVSADM_REP);

    if (dir != NULL) {
      tmp2 = xmalloc (strlen (dir) + sizeof (CVSADM_REP_REMOTE) + 10);
      (void) sprintf (tmp2, "%s/%s", dir, CVSADM_REP_REMOTE);
    } else {
      tmp2 = xstrdup (CVSADM_REP_REMOTE);
    }

    /* We try to read remote repository first, if we are handling remotes */
    if (handling_remotes) {
      fpin = CVS_FOPEN (tmp2, "r");

      if (fpin != NULL) {
        getline (&repos, &repos_allocated, fpin);
        fclose (fpin);
      }
    }

    if (repos_allocated == 0) {
    /*
     * The assumption here is that the repository is always contained in the
     * first line of the "Repository" file.
     */
    fpin = CVS_FOPEN (tmp, "r");

    if (fpin == NULL)
    {
        int save_errno = errno;
        char *cvsadm;

        if (dir != NULL)
        {
            cvsadm = xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
            (void) sprintf (cvsadm, "%s/%s", dir, CVSADM);
        }
        else
            cvsadm = xstrdup (CVSADM);

        if (!isdir (cvsadm))
        {
            error (0, 0, "in directory %s:", xupdate_dir);
            error (1, 0, "there is no version here; do '%s checkout' first",
```

```
                program_name);
        }
        free (cvsadm);

        if (existence_error (save_errno))
        {
            /* FIXME: This is a very poorly worded error message.  It
               occurs at least in the case where the user manually
               creates a directory named CVS, so the error message
               should be more along the lines of "CVS directory found
               without administrative files; use CVS to create the CVS
               directory, or rename it to something else if the
               intention is to store something besides CVS
               administrative files".  */
            error (0, 0, "in directory %s:", xupdate_dir);
            error (1, 0, "*PANIC* administration files missing");
        }

        error (1, save_errno, "cannot open %s", tmp);
    }

    if (getline (&repos, &repos_allocated, fpin) < 0)
    {
        /* FIXME: should be checking for end of file separately.  */
        error (0, 0, "in directory %s:", xupdate_dir);
        error (1, errno, "cannot read %s", CVSADM_REP);
    }
    if (fclose (fpin) < 0)
        error (0, errno, "cannot close %s", tmp);
    free (tmp);
    }

    if ((cp = strrchr (repos, '\n')) != NULL)
        *cp = '\0';                 /* strip the newline */

    /*
     * If this is a relative repository pathname, turn it into an absolute
     * one by tacking on the CVSROOT environment variable. If the CVSROOT
     * environment variable is not set, die now.
     */
    if (strcmp (repos, ". .") == 0 || strncmp (repos, ". ./", 3) == 0)
    {
        error (0, 0, "in directory %s:", xupdate_dir);
        error (0, 0, "`. .'-relative repositories are not supported.");
        error (1, 0, "illegal source repository");
    }
    if (! isabsolute(repos))
    {
        char *newrepos;

        if (CVSroot_original == NULL)
        {
            error (0, 0, "in directory %s:", xupdate_dir);
            error (0, 0, "must set the CVSROOT environment variable\n");
            error (0, 0, "or specify the '-d' option to %s.", program_name);
            error (1, 0, "illegal repository setting");
        }
        newrepos = xmalloc (strlen (CVSroot_directory) + strlen (repos) + 10);
        (void) sprintf (newrepos, "%s/%s", CVSroot_directory, repos);
        free (repos);
        repos = newrepos;
    }

    Sanitize_Repository_Name (repos);

    return repos;
}

/*
 * Return a pointer to the repository name relative to CVSROOT from a
 * possibly fully qualified repository
 */
char *
Short_Repository (repository)
    char *repository;
{
    if (repository == NULL)
        return (NULL);

    /* If repository matches CVSroot at the beginning, strip off CVSroot */
    /* And skip leading '/' in rep, in case CVSroot ended with '/'. */
    if (strncmp (CVSroot_directory, repository,
                 strlen (CVSroot_directory)) == 0)
    {
        char *rep = repository + strlen (CVSroot_directory);
        return (*rep == '/') ? rep+1 : rep;
    }
    else
        return (repository);
}
```

```
/* Sanitize the repository name (in place) by removing trailing
 * slashes and a trailing "." if present.  It should be safe for
 * callers to use strcat and friends to create repository names.
 * Without this check, names like "/path/to/repos/./foo" and
 * "/path/to/repos//foo" would be created.  For example, one
 * significant case is the CVSROOT-detection code in commit.c.  It
 * decides whether or not it needs to rebuild the administrative file
 * database by doing a string compare.  If we've done a 'cvs co .' to
 * get the CVSROOT files, "/path/to/repos/./CVSROOT" and
 * "/path/to/repos/CVSROOT" are the arguments that are compared!
 *
 * This function ends up being called from the same places as
 * strip_path, though what it does is much more conservative.  Many
 * comments about this operation (which was scattered around in
 * several places in the source code) ran thus:
 *
 *    ``repository ends with "/."; omit it.  This sort of thing used
 *    to be taken care of by strip_path.  Now we try to be more
 *    selective.  I suspect that it would be even better to push it
 *    back further someday, so that the trailing "/." doesn't get into
 *    repository in the first place, but we haven't taken things that
 *    far yet."          -Jim Kingdon (recurse.c, 07-Sep-97)
 *
 * Ahh, all too true.  The major consideration is RELATIVE_REPOS.  If
 * the "/." doesn't end up in the repository while RELATIVE_REPOS is
 * defined, there will be nothing in the CVS/Repository file.  I
 * haven't verified that the remote protocol will handle that
 * correctly yet, so I've not made that change. */

void
Sanitize_Repository_Name (repository)
    char *repository;
{
    size_t len;

    assert (repository != NULL);

    strip_trailing_slashes (repository);

    len = strlen (repository);
    if (len >= 2
        && repository[len - 1] == '.'
        && ISDIRSEP (repository[len - 2]))
    {
        repository[len - 2] = '\0';
    }
}
```

A.51 root.c

```
/*
 * Copyright (c) 1992, Mark D. Baushke
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Name of Root
 *
 * Determine the path to the CVSROOT and set "Root" accordingly.
 * If this looks like of modified clone of Name_Repository() in
 * repos.c, it is. . .
 */

#include "cvs.h"
#include "getline.h"

/* Printable names for things in the CVSroot_method enum variable.
   Watch out if the enum is changed in cvs.h! */

char *method_names[] = {
    "local", "server (rsh)", "pserver", "kserver", "gserver", "ext"
};

#ifndef DEBUG

char *
Name_Root (dir, update_dir)
    char *dir;
    char *update_dir;
{
    FILE *fpin;
    char *ret, *xupdate_dir;
    char *root = NULL;
    size_t root_allocated = 0;
    char *tmp;
    char *cvsadm;
    char *cp;

    if (update_dir && *update_dir)
        xupdate_dir = update_dir;
    else
        xupdate_dir = ".";

    if (dir != NULL)
    {
        cvsadm = xmalloc (strlen (dir) + sizeof (CVSADM) + 10);
        (void) sprintf (cvsadm, "%s/%s", dir, CVSADM);
        tmp = xmalloc (strlen (dir) + sizeof (CVSADM_ROOT) + 10);
        (void) sprintf (tmp, "%s/%s", dir, CVSADM_ROOT);
    }
    else
    {
        cvsadm = xstrdup (CVSADM);
        tmp = xstrdup (CVSADM_ROOT);
    }

    /*
     * Do not bother looking for a readable file if there is no cvsadm
     * directory present.
     *
     * It is possible that not all repositories will have a CVS/Root
     * file. This is ok, but the user will need to specify -d
     * /path/name or have the environment variable CVSROOT set in
     * order to continue.  */
    if ((!isdir (cvsadm)) || (!isreadable (tmp)))
    {
        ret = NULL;
        goto out;
    }

    /*
     * The assumption here is that the CVS Root is always contained in the
     * first line of the "Root" file.
     */
    fpin = open_file (tmp, "r");

    if (getline (&root, &root_allocated, fpin) < 0)
    {
        /* FIXME: should be checking for end of file separately; errno
           is not set in that case.  */
        error (0, 0, "in directory %s:", xupdate_dir);
        error (0, errno, "cannot read %s", CVSADM_ROOT);
        error (0, 0, "please correct this problem");
        ret = NULL;
        goto out;
    }
    (void) fclose (fpin);
    if ((cp = strrchr (root, '\n')) != NULL)
```

```
        *cp = '\0';                 /* strip the newline */
90
    /*
     * root now contains a candidate for CVSroot. It must be an
     * absolute pathname or specify a remote server.
     */

    if (
#ifdef CLIENT_SUPPORT
        (strchr (root, ':') == NULL) &&
#endif
100         ! isabsolute (root))
    {
        error (0, 0, "in directory %s:", xupdate_dir);
        error (0, 0,
               "ignoring %s because it does not contain an absolute pathname.",
               CVSADM_ROOT);
        ret = NULL;
        goto out;
    }

110 #ifdef CLIENT_SUPPORT
    if ((strchr (root, ':') == NULL) && !isdir (root))
#else /* ! CLIENT_SUPPORT */
    if (!isdir (root))
#endif /* CLIENT_SUPPORT */
    {
        error (0, 0, "in directory %s:", xupdate_dir);
        error (0, 0,
               "ignoring %s because it specifies a non-existent repository %s",
               CVSADM_ROOT, root);
120         ret = NULL;
        goto out;
    }

    /* allocate space to return and fill it in */
    strip_trailing_slashes (root);
    ret = xstrdup (root);
 out:
    free (cvsadm);
    free (tmp);
130 if (root != NULL)
        free (root);
    return (ret);
}

/*
 * Write the CVS/Root file so that the environment variable CVSROOT
 * and/or the -d option to cvs will be validated or not necessary for
 * future work.
 */
140 void
Create_Root (dir, rootdir)
    char *dir;
    char *rootdir;
{
    FILE *fout;
    char *tmp;

    if (noexec)
        return;
150
    /* record the current cvs root */

    if (rootdir != NULL)
    {
        if (dir != NULL)
        {
            tmp = xmalloc (strlen (dir) + sizeof (CVSADM_ROOT) + 10);
            (void) sprintf (tmp, "%s/%s", dir, CVSADM_ROOT);
        }
160     else
            tmp = xstrdup (CVSADM_ROOT);

        fout = open_file (tmp, "w+");
        if (fprintf (fout, "%s\n", rootdir) < 0)
            error (1, errno, "write to %s failed", tmp);
        if (fclose (fout) == EOF)
            error (1, errno, "cannot close %s", tmp);
        free (tmp);
    }
170 }

#endif /* ! DEBUG */


/* The root_allow_* stuff maintains a list of legal CVSROOT
   directories.  Then we can check against them when a remote user
   hands us a CVSROOT directory.  */
```

```
static unsigned int root_allow_count;
static char **root_allow_vector;
static unsigned int root_allow_size;

void
root_allow_add (arg)
    char *arg;
{
    char *p;

    if (root_allow_size <= root_allow_count)
    {
        if (root_allow_size == 0)
        {
            root_allow_size = 1;
            root_allow_vector =
                (char **) malloc (root_allow_size * sizeof (char *));
        }
        else
        {
            root_allow_size *= 2;
            root_allow_vector =
                (char **) realloc (root_allow_vector,
                                   root_allow_size * sizeof (char *));
        }

        if (root_allow_vector == NULL)
        {
        no_memory:
            /* Strictly speaking, we're not supposed to output anything
               now.  But we're about to exit(), give it a try.  */
            printf ("E Fatal server error, aborting.\n\
error ENOMEM Virtual memory exhausted.\n");

            /* I'm doing this manually rather than via error_exit ()
               because I'm not sure whether we want to call server_cleanup.
               Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
            /* Hook for OS-specific behavior, for example socket
               subsystems on NT and OS2 or dealing with windows
               and arguments on Mac.  */
            SYSTEM_CLEANUP ();
#endif

            exit (EXIT_FAILURE);
        }
    }
    p = malloc (strlen (arg) + 1);
    if (p == NULL)
        goto no_memory;
    strcpy (p, arg);
    root_allow_vector[root_allow_count++] = p;
}

void
root_allow_free ()
{
    if (root_allow_vector != NULL)
        free (root_allow_vector);
    root_allow_count = 0;
    root_allow_size = 0;
}

int
root_allow_ok (arg)
    char *arg;
{
    unsigned int i;

    if (root_allow_count == 0)
    {
        /* Probably someone upgraded from CVS before 1.9.10 to 1.9.10
           or later without reading the documentation about
           -allow-root.  Printing an error here doesn't disclose any
           particularly useful information to an attacker because a
           CVS server configured in this way won't let *anyone* in.  */

        /* Note that we are called from a context where we can spit
           back "error" rather than waiting for the next request which
           expects responses.  */
        printf ("\
error 0 Server configuration missing --allow-root in inetd.conf\n");
        error_exit ();
    }

    for (i = 0; i < root_allow_count; ++i)
        if (strcmp (root_allow_vector[i], arg) == 0)
            return 1;
    return 0;
```

```
}

/* Parse a CVSROOT variable into its constituent parts – method,
 * username, hostname, directory.  The prototypical CVSROOT variable
 * looks like:
 *
 * :method:userhost:path
 *
 * Some methods may omit fields; local, for example, doesn't need user
 * and host.
 *
 * Returns zero on success, non-zero on failure. */

char *CVSroot_original = NULL; /* the CVSroot that was passed in */
int client_active;        /* nonzero if we are doing remote access */
CVSmethod CVSroot_method; /* one of the enum values defined in cvs.h */
char *CVSroot_username;   /* the username or NULL if method == local */
char *CVSroot_hostname;   /* the hostname or NULL if method == local */
char *CVSroot_directory;  /* the directory name */

#ifdef AUTH_SERVER_SUPPORT
/* Die if CVSroot_directory and Pserver_Repos don't*/
static void
check_root_consistent ()
{
    /* FIXME: Should be using a deferred error, as the rest of
       serve_root does.  As it is now the call to error could conceivably
       cause deadlock, as noted in server_cleanup.  Best solution would
       presumably be to write some code so that error() automatically
       defers the error in those cases where that is needed.  */
    /* FIXME?  Possible that the wording should be more clear (e.g.
         Root says "%s" but pserver protocol says "%s"
       or something which would aid people who are writing implementations
       of the client side of the CVS protocol.  I don't see any security
       problem with revealing that information.  */
    if ((Pserver_Repos != NULL) && (CVSroot_directory != NULL))
        if (strcmp (Pserver_Repos, CVSroot_directory) != 0)
            error (1, 0, "repository mismatch: \"%s\" vs \"%s\"",
                   Pserver_Repos, CVSroot_directory);
}

#endif /* AUTH_SERVER_SUPPORT */


int
parse_cvsroot (CVSroot)
    char *CVSroot;
{
    static int cvsroot_parsed = 0;
    char *cvsroot_copy, *p;
    int check_hostname;

    /* Don't go through the trouble twice. */
    /*    if (cvsroot_parsed)
    {
        error (0, 0, "WARNING (parse_cvsroot): someone called me twice!\n");
        return 0;
    }
    */

    CVSroot_original = xstrdup (CVSroot);
    cvsroot_copy = xstrdup (CVSroot);

    if ((*cvsroot_copy == ':'))
    {
        char *method = ++cvsroot_copy;

        /* Access method specified, as in
         * "cvs -d :pserver:userhost:/path",
         * "cvs -d :local:e:\path", or
         * "cvs -d :kserver:userhost:/path".
         * We need to get past that part of CVSroot before parsing the
         * rest of it.
         */

        if (! (p = strchr (method, ':')))
        {
            error (0, 0, "bad CVSroot: %s", CVSroot);
            return 1;
        }
        *p = '\0';
        cvsroot_copy = ++p;

        /* Now we have an access method – see if it's valid. */

        if (strcmp (method, "local") == 0)
            CVSroot_method = local_method;
        else if (strcmp (method, "pserver") == 0)
            CVSroot_method = pserver_method;
```

```
        else if (strcmp (method, "kserver") == 0)
            CVSroot_method = kserver_method;
        else if (strcmp (method, "gserver") == 0)
            CVSroot_method = gserver_method;
        else if (strcmp (method, "server") == 0)
            CVSroot_method = server_method;
        else if (strcmp (method, "ext") == 0)
            CVSroot_method = ext_method;
        else
        {
            error (0, 0, "unknown method in CVSroot: %s", CVSroot);
            return 1;
        }
    }
    else
    {
        /* If the method isn't specified, assume
           SERVER_METHOD/EXT_METHOD if the string contains a colon or
           LOCAL_METHOD otherwise.  */

        CVSroot_method = ((strchr (cvsroot_copy, ':'))
#ifdef RSH_NOT_TRANSPARENT
                          ? server_method
#else
                          ? ext_method
#endif
                          : local_method);
    }

    client_active = (CVSroot_method != local_method);

    /* Check for username/hostname if we're not LOCAL_METHOD. */

    CVSroot_username = NULL;
    CVSroot_hostname = NULL;

    if (CVSroot_method != local_method)
    {
        /* Check to see if there is a username in the string. */

        if ((p = strchr (cvsroot_copy, '@')))
        {
            CVSroot_username = cvsroot_copy;
            *p = '\0';
            cvsroot_copy = ++p;
            if (*CVSroot_username == '\0')
                CVSroot_username = NULL;
        }

        if ((p = strchr (cvsroot_copy, ':')))
        {
            CVSroot_hostname = cvsroot_copy;
            *p = '\0';
            cvsroot_copy = ++p;

            if (*CVSroot_hostname == '\0')
                CVSroot_hostname = NULL;
        }
    }

    CVSroot_directory = cvsroot_copy;
#ifdef AUTH_SERVER_SUPPORT
    check_root_consistent ();
#endif /* AUTH_SERVER_SUPPORT */

#if ! defined (CLIENT_SUPPORT) && ! defined (DEBUG)
    if (CVSroot_method != local_method)
    {
        error (0, 0, "Your CVSROOT is set for a remote access method");
        error (0, 0, "but your CVS executable doesn't support it");
        error (0, 0, "(%s)", CVSroot);
        return 1;
    }
#endif

    /* Do various sanity checks. */

    if (CVSroot_username && ! CVSroot_hostname)
    {
        error (0, 0, "missing hostname in CVSROOT: %s", CVSroot);
        return 1;
    }

    check_hostname = 0;
    switch (CVSroot_method)
    {
    case local_method:
        if (CVSroot_username || CVSroot_hostname)
        {
            error (0, 0, "can't specify hostname and username in CVSROOT");
```

```
            error (0, 0, "when using local access method");
            error (0, 0, "(%s)", CVSroot);
            return 1;
        }
        /* cvs.texinfo has always told people that CVSROOT must be an
           absolute pathname.  Furthermore, attempts to use a relative
           pathname produced various errors (I couldn't get it to work),
           so there would seem to be little risk in making this a fatal
           error.  */
        if (!isabsolute (CVSroot_directory))
            error (1, 0, "CVSROOT %s must be an absolute pathname",
                   CVSroot_directory);
        break;
    case kserver_method:
#ifndef HAVE_KERBEROS
        error (0, 0, "Your CVSROOT is set for a kerberos access method");
        error (0, 0, "but your CVS executable doesn't support it");
        error (0, 0, "(%s)", CVSroot);
        return 1;
#endif
        check_hostname = 1;
        break;
    case gserver_method:
#ifndef HAVE_GSSAPI
        error (0, 0, "Your CVSROOT is set for a GSSAPI access method");
        error (0, 0, "but your CVS executable doesn't support it");
        error (0, 0, "(%s)", CVSroot);
        return 1;
#endif
        check_hostname = 1;
        break;
    case server_method:
    case ext_method:
    case pserver_method:
        check_hostname = 1;
        break;
    }

    if (check_hostname)
    {
        if (! CVSroot_hostname)
        {
            error (0, 0, "didn't specify hostname in CVSROOT: %s", CVSroot);
            return 1;
        }
    }

    if (*CVSroot_directory == '\0')
    {
        error (0, 0, "missing directory in CVSROOT: %s", CVSroot);
        return 1;
    }

    /* Hooray!  We finally parsed it! */
    return 0;
}


/* Set up the global CVSroot* variables as if we're using the local
   repository DIR. */

void
set_local_cvsroot (dir)
    char *dir;
{
    CVSroot_original = xstrdup (dir);
    CVSroot_method = local_method;
    CVSroot_directory = CVSroot_original;
#ifdef AUTH_SERVER_SUPPORT
    check_root_consistent ();
#endif /* AUTH_SERVER_SUPPORT */
    CVSroot_username = NULL;
    CVSroot_hostname = NULL;
    client_active = 0;
}


#ifdef DEBUG
/* This is for testing the parsing function. */

#include <stdio.h>

char *CVSroot;
char *program_name = "testing";
char *command_name = "parse_cvsroot"; /* XXX is this used??? */

void
main (argc, argv)
    int argc;
    char *argv[];
```

```
{
    program_name = argv[0];

    if (argc != 2)
    {
        fprintf (stderr, "Usage: %s <CVSROOT>\n", program_name);
        exit (2);
    }

    if (parse_cvsroot (argv[1]))
    {
        fprintf (stderr, "%s: Parsing failed.", program_name);
        exit (1);
    }
    printf ("CVSroot: %s\n", argv[1]);
    printf ("CVSroot_method: %s\n", method_names[CVSroot_method]);
    printf ("CVSroot_username: %s\n",
            CVSroot_username ? CVSroot_username : "NULL");
    printf ("CVSroot_hostname: %s\n",
            CVSroot_hostname ? CVSroot_hostname : "NULL");
    printf ("CVSroot_directory: %s\n", CVSroot_directory);

  exit (0);
  /* NOTREACHED */
}
#endif
```

A.52 rtag.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Rtag
 *
 * Add or delete a symbolic name to an RCS file, or a collection of RCS files.
 * Uses the modules database, if necessary.
 */

#include "cvs.h"

static int check_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int check_filesdoneproc PROTO ((void *callerdat, int err,
                                       char *repos, char *update_dir,
                                       List *entries));
static int pretag_proc PROTO((char *repository, char *filter));
static void masterlist_delproc PROTO((Node *p));
static void tag_delproc PROTO((Node *p));
static int pretag_list_proc PROTO((Node *p, void *closure));

static Dtype rtag_dirproc PROTO ((void *callerdat, char *dir,
                                  char *repos, char *update_dir,
                                  List *entries));
static int rtag_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int rtag_filesdoneproc PROTO ((void *callerdat, int err,
                                      char *repos, char *update_dir,
                                      List *entries));
static int rtag_proc PROTO((int *pargc, char **argv, char *xwhere,
                            char *mwhere, char *mfile, int shorten,
                            int local_specified, char *mname, char *msg));
static int rtag_delete PROTO((RCSNode *rcsfile));


struct tag_info
{
    Ctype status;
    char *rev;
    char *tag;
    char *options;
};

struct master_lists
{
    List *tlist;
};

static List *mtlist;
static List *tlist;

static char *symtag;
static char *numtag;
static int numtag_validated = 0;
static int delete_flag;          /* adding a tag by default */
static int attic_too;            /* remove tag from Attic files */
static int branch_mode;          /* make an automagic "branch" tag */
static char *date;
static int local;                /* recursive by default */
static int force_tag_match = 1; /* force by default */
static int force_tag_move;              /* don't move existing tags by default */

static const char *const rtag_usage[] =
{
    "Usage: %s %s [-aflRnF] [-b] [-d] [-r tag|-D date] tag modules. . .\n",
    "\t-a\tClear tag from removed files that would not otherwise be tagged.\n",
    "\t-f\tForce a head revision match if tag/date not found.\n",
    "\t-l\tLocal directory only, not recursive\n",
    "\t-R\tProcess directories recursively.\n",
    "\t-n\tNo execution of 'tag program'\n",
    "\t-d\tDelete the given Tag.\n",
    "\t-b\tMake the tag a \"branch\" tag, allowing concurrent development.\n",
    "\t-r rev\tExisting revision/tag.\n",
    "\t-D\tExisting date.\n",
    "\t-F\tMove tag if it already exists\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
rtag (argc, argv)
    int argc;
    char **argv;
{
    register int i;
    int c;
```

```
    DBM *db;
    int run_module_prog = 1;
    int err = 0;

    if (argc == -1)
        usage (rtag_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+FanfQqlRdbr:D:")) != -1)
    {
        switch (c)
        {
            case 'a':
                attic_too = 1;
                break;
            case 'n':
                run_module_prog = 0;
                break;
            case 'Q':
            case 'q':
#ifdef SERVER_SUPPORT
                /* The CVS 1.5 client sends these options (in addition to
                   Global_option requests), so we must ignore them. */
                if (!server_active)
#endif
                    error (1, 0,
                           "-q or -Q must be specified before \"%s\"",
                           command_name);
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'd':
                delete_flag = 1;
                break;
            case 'f':
                force_tag_match = 0;
                break;
            case 'b':
                branch_mode = 1;
                break;
            case 'r':
                numtag = optarg;
                break;
            case 'D':
                if (date)
                    free (date);
                date = Make_Date (optarg);
                break;
            case 'F':
                force_tag_move = 1;
                break;
            case '?':
            default:
                usage (rtag_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;
    if (argc < 2)
        usage (rtag_usage);
    symtag = argv[0];
    argc--;
    argv++;

    if (date && numtag)
        error (1, 0, "-r and -D options are mutually exclusive");
    if (delete_flag && branch_mode)
        error (0, 0, "warning: -b ignored with -d options");
    RCS_check_tag (symtag);

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        /* We're the client side.  Fire up the remote server. */
        start_server ();

        ign_setup ();

        if (!force_tag_match)
            send_arg ("-f");
        if (local)
            send_arg("-l");
        if (delete_flag)
            send_arg("-d");
        if (branch_mode)
```

```
            send_arg("-b");
        if (force_tag_move)
            send_arg("-F");
        if (!run_module_prog)
            send_arg("-n");
        if (attic_too)
            send_arg("-a");

        if (numtag)
            option_with_arg ("-r", numtag);
        if (date)
            client_senddate (date);

        send_arg (symtag);

        {
            int i;
            for (i = 0; i < argc; ++i)
                send_arg (argv[i]);
        }

        send_to_server ("rtag\012", 0);
        return get_responses_and_close ();
    }
#endif

    db = open_module ();
    for (i = 0; i < argc; i++)
    {
        /* XXX last arg should be repository, but doesn't make sense here */
        history_write ('T', (delete_flag ? "D" : (numtag ? numtag :
                    (date ? date : "A"))), symtag, argv[i], "");
        err += do_module (db, argv[i], TAG,
                          delete_flag ? "Untagging" : "Tagging",
                          rtag_proc, (char *) NULL, 0, 0, run_module_prog,
                          symtag);
    }
    close_module (db);
    return (err);
}

/*
 * callback proc for doing the real work of tagging
 */
/* ARGSUSED */
static int
rtag_proc (pargc, argv, xwhere, mwhere, mfile, shorten, local_specified,
           mname, msg)
    int *pargc;
    char **argv;
    char *xwhere;
    char *mwhere;
    char *mfile;
    int shorten;
    int local_specified;
    char *mname;
    char *msg;
{
    /* Begin section which is identical to patch_proc--should this
       be abstracted out somehow?  */
    int err = 0;
    int which;
    char *repository;
    char *where;

    repository = xmalloc (strlen (CVSroot_directory) + strlen (argv[0])
                          + (mfile == NULL ? 0 : strlen (mfile)) + 30);
    (void) sprintf (repository, "%s/%s", CVSroot_directory, argv[0]);
    where = xmalloc (strlen (argv[0]) + (mfile == NULL ? 0 : strlen (mfile))
                     + 10);
    (void) strcpy (where, argv[0]);

    /* if mfile isn't null, we need to set up to do only part of the module */
    if (mfile != NULL)
    {
        char *cp;
        char *path;

        /* if the portion of the module is a path, put the dir part on repos */
        if ((cp = strrchr (mfile, '/')) != NULL)
        {
            *cp = '\0';
            (void) strcat (repository, "/");
            (void) strcat (repository, mfile);
            (void) strcat (where, "/");
            (void) strcat (where, mfile);
            mfile = cp + 1;
        }

        /* take care of the rest */
```

```
        path = xmalloc (strlen (repository) + strlen (mfile) + 5);
        (void) sprintf (path, "%s/%s", repository, mfile);
        if (isdir (path))
        {
            /* directory means repository gets the dir tacked on */
            (void) strcpy (repository, path);
            (void) strcat (where, "/");
            (void) strcat (where, mfile);
        }
        else
        {
            int i;

            /* a file means muck argv */
            for (i = 1; i < *pargc; i++)
                free (argv[i]);
            argv[1] = xstrdup (mfile);
            (*pargc) = 2;
        }
        free (path);
    }

    /* cd to the starting repository */
    if ( CVS_CHDIR (repository) < 0)
    {
        error (0, errno, "cannot chdir to %s", repository);
        free (repository);
        return (1);
    }
    free (repository);
    /* End section which is identical to patch_proc.  */

    if (delete_flag || attic_too || (force_tag_match && numtag))
        which = W_REPOS | W_ATTIC;
    else
        which = W_REPOS;

    if (numtag != NULL && !numtag_validated)
    {
        tag_check_valid (numtag, *pargc - 1, argv + 1, local, 0, NULL);
        numtag_validated = 1;
    }

    /* check to make sure they are authorized to tag all the
       specified files in the repository */

    mtlist = getlist();
    err = start_recursion (check_fileproc, check_filesdoneproc,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                           *pargc - 1, argv + 1, local, which, 0, 1,
                           where, 1);

    if (err)
    {
       error (1, 0, "correct the above errors first!");
    }

    /* start the recursion processor */
    err = start_recursion (rtag_fileproc, rtag_filesdoneproc, rtag_dirproc,
                           (DIRLEAVEPROC) NULL, NULL,
                           *pargc - 1, argv + 1, local,
                           which, 0, 0, where, 1);
    free (where);
    dellist(&mtlist);

    return (err);
}

/* check file that is to be tagged */
/* All we do here is add it to our list */

static int
check_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *xdir;
    Node *p;
    Vers_TS *vers;

    if (finfo->update_dir[0] == '\0')
        xdir = ".";
    else
        xdir = finfo->update_dir;
    if ((p = findnode (mtlist, xdir)) != NULL)
    {
        tlist = ((struct master_lists *) p->data)->tlist;
    }
    else
    {
```

```
        struct master_lists *ml;

        tlist = getlist ();
        p = getnode ();
        p->key = xstrdup (xdir);
        p->type = UPDATE;
        ml = (struct master_lists *)
            xmalloc (sizeof (struct master_lists));
        ml->tlist = tlist;
        p->data = (char *) ml;
        p->delproc = masterlist_delproc;
        (void) addnode (mtlist, p);
    }
    /* do tlist */
    p = getnode ();
    p->key = xstrdup (finfo->file);
    p->type = UPDATE;
    p->delproc = tag_delproc;
    vers = Version_TS (finfo, NULL, NULL, NULL, 0, 0);
    p->data = RCS_getversion(vers->srcfile, numtag, date, force_tag_match,
                             (int *) NULL);
    if (p->data != NULL)
    {
        int addit = 1;
        char *oversion;

        oversion = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1,
                                   (int *) NULL);
        if (oversion == NULL)
        {
            if (delete_flag)
            {
                addit = 0;
            }
        }
        else if (strcmp(oversion, p->data) == 0)
        {
            addit = 0;
        }
        else if (!force_tag_move)
        {
            addit = 0;
        }
        if (oversion != NULL)
        {
            free(oversion);
        }
        if (!addit)
        {
            free(p->data);
            p->data = NULL;
        }
    }
    freevers_ts (&vers);
    (void) addnode (tlist, p);
    return (0);
}

static int
check_filesdoneproc (callerdat, err, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
{
    int n;
    Node *p;

    p = findnode(mtlist, update_dir);
    if (p != NULL)
    {
        tlist = ((struct master_lists *) p->data)->tlist;
    }
    else
    {
        tlist = (List *) NULL;
    }
    if ((tlist == NULL) || (tlist->list->next == tlist->list))
    {
        return (err);
    }
    if ((n = Parse_Info(CVSROOTADM_TAGINFO, repos, pretag_proc, 1)) > 0)
    {
        error (0, 0, "Pre-tag check failed");
        err += n;
    }
    return (err);
}
```

```
static int
pretag_proc(repository, filter)
    char *repository;
    char *filter;
{
    if (filter[0] == '/')
    {
        char *s, *cp;

        s = xstrdup(filter);
        for (cp=s; *cp; cp++)
        {
            if (isspace(*cp))
            {
                *cp = '\0';
                break;
            }
        }
        if (!isfile(s))
        {
            error (0, errno, "cannot find pre-tag filter '%s'", s);
            free(s);
            return (1);
        }
        free(s);
    }
    run_setup (filter);
    run_arg (symtag);
    run_arg (delete_flag ? "del" : force_tag_move ? "mov" : "add");
    run_arg (repository);
    walklist(tlist, pretag_list_proc, NULL);
    return (run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL));
}

static void
masterlist_delproc(p)
    Node *p;
{
    struct master_lists *ml;

    ml = (struct master_lists *)p->data;
    dellist(&ml->tlist);
    free(ml);
    return;
}

static void
tag_delproc(p)
    Node *p;
{
    if (p->data != NULL)
    {
        free(p->data);
        p->data = NULL;
    }
    return;
}

static int
pretag_list_proc(p, closure)
    Node *p;
    void *closure;
{
    if (p->data != NULL)
    {
        run_arg(p->key);
        run_arg(p->data);
    }
    return (0);
}

/*
 * Called to tag a particular file, as appropriate with the options that were
 * set above.
 */
/* ARGSUSED */
static int
rtag_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    RCSNode *rcsfile;
    char *version, *rev;
    int retcode = 0;

    /* Lock the directory if it is not already locked.  We might be
       able to rely on rtag_dirproc for this.  */

    /* It would be nice to provide consistency with respect to
       commits; however CVS lacks the infrastructure to do that (see
```

```
    Concurrency in cvs.texinfo and comment in do_recursion).  We
    can and will prevent simultaneous tag operations from
    interfering with each other, by write locking each directory as
    we enter it, and unlocking it as we leave it.  */

lock_dir_for_write (finfo->repository);

/* find the parsed RCS data */
if ((rcsfile = finfo->rcs) == NULL)
    return (1);

/*
 * For tagging an RCS file which is a symbolic link, you'd best be
 * running with RCS 5.6, since it knows how to handle symbolic links
 * correctly without breaking your link!
 */

if (delete_flag)
    return (rtag_delete (rcsfile));

/*
 * If we get here, we are adding a tag.  But, if -a was specified, we
 * need to check to see if a -r or -D option was specified.  If neither
 * was specified and the file is in the Attic, remove the tag.
 */
if (attic_too && (!numtag && !date))
{
    if ((rcsfile->flags & VALID) && (rcsfile->flags & INATTIC))
        return (rtag_delete (rcsfile));
}

version = RCS_getversion (rcsfile, numtag, date, force_tag_match,
                          (int *) NULL);
if (version == NULL)
{
    /* If -a specified, clean up any old tags */
    if (attic_too)
        (void) rtag_delete (rcsfile);

    if (!quiet && !force_tag_match)
    {
        error (0, 0, "cannot find tag `%s' in `%s'",
               numtag ? numtag : "head", rcsfile->path);
        return (1);
    }
    return (0);
}
if (numtag && isdigit (*numtag) && strcmp (numtag, version) != 0)
{

    /*
     * We didn't find a match for the numeric tag that was specified, but
     * that's OK.  just pass the numeric tag on to rcs, to be tagged as
     * specified.  Could get here if one tried to tag "1.1.1" and there
     * was a 1.1.1 branch with some head revision.  In this case, we want
     * the tag to reference "1.1.1" and not the revision at the head of
     * the branch.  Use a symbolic tag for that.
     */
    rev = branch_mode ? RCS_magicrev (rcsfile, version) : numtag;
    retcode = RCS_settag(rcsfile, symtag, numtag);
    if (retcode == 0)
        RCS_rewrite (rcsfile, NULL, NULL);
}
else
{
    char *oversion;

    /*
     * As an enhancement for the case where a tag is being re-applied to
     * a large body of a module, make one extra call to RCS_getversion to
     * see if the tag is already set in the RCS file.  If so, check to
     * see if it needs to be moved.  If not, do nothing.  This will
     * likely save a lot of time when simply moving the tag to the
     * "current" head revisions of a module - which I have found to be a
     * typical tagging operation.
     */
    rev = branch_mode ? RCS_magicrev (rcsfile, version) : version;
    oversion = RCS_getversion (rcsfile, symtag, (char *) NULL, 1,
                               (int *) NULL);
    if (oversion != NULL)
    {
        int isbranch = RCS_nodeisbranch (finfo->rcs, symtag);

        /*
         * if versions the same and neither old or new are branches don't
         * have to do anything
         */
        if (strcmp (version, oversion) == 0 && !branch_mode && !isbranch)
        {
            free (oversion);
```

```
                free (version);
                return (0);
            }

            if (!force_tag_move)
            {
                /* we're NOT going to move the tag */
                (void) printf ("W %s", finfo->fullname);

                (void) printf (" : %s already exists on %s %s",
                               symtag, isbranch ? "branch" : "version",
                               oversion);
                (void) printf (" : NOT MOVING tag to %s %s\n",
                               branch_mode ? "branch" : "version", rev);
                free (oversion);
                free (version);
                return (0);
            }
            free (oversion);
        }
        retcode = RCS_settag(rcsfile, symtag, rev);
        if (retcode == 0)
            RCS_rewrite (rcsfile, NULL, NULL);
    }

    if (retcode != 0)
    {
        error (1, retcode == -1 ? errno : 0,
               "failed to set tag `%s' to revision `%s' in `%s'",
               symtag, rev, rcsfile->path);
        if (branch_mode)
            free (rev);
        free (version);
        return (1);
    }
    if (branch_mode)
        free (rev);
    free (version);
    return (0);
}

/*
 * If -d is specified, "force_tag_match" is set, so that this call to
 * RCS_getversion() will return a NULL version string if the symbolic
 * tag does not exist in the RCS file.
 *
 * If the -r flag was used, numtag is set, and we only delete the
 * symtag from files that have numtag.
 *
 * This is done here because it's MUCH faster than just blindly calling
 * "rcs" to remove the tag... trust me.
 */
static int
rtag_delete (rcsfile)
    RCSNode *rcsfile;
{
    char *version;
    int retcode;

    if (numtag)
    {
        version = RCS_getversion (rcsfile, numtag, (char *) NULL, 1,
                                  (int *) NULL);
        if (version == NULL)
            return (0);
        free (version);
    }

    version = RCS_getversion (rcsfile, symtag, (char *) NULL, 1,
                              (int *) NULL);
    if (version == NULL)
        return (0);
    free (version);

    if ((retcode = RCS_deltag(rcsfile, symtag)) != 0)
    {
        if (!quiet)
            error (0, retcode == -1 ? errno : 0,
                   "failed to remove tag `%s' from `%s'", symtag,
                   rcsfile->path);
        return (1);
    }
    RCS_rewrite (rcsfile, NULL, NULL);
    return (0);
}

/* Clear any lock we may hold on the current directory.  */

static int
rtag_filesdoneproc (callerdat, err, repos, update_dir, entries)
```

```
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
{
    Lock_Cleanup ();

    return (err);
}

/*
 * Print a warm fuzzy message
 */
/* ARGSUSED */
static Dtype
rtag_dirproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    if (ignore_directory (update_dir))
    {
        /* print the warm fuzzy message */
        if (!quiet)
          error (0, 0, "Ignoring %s", update_dir);
        return R_SKIP_ALL;
    }

    if (!quiet)
        error (0, 0, "%s %s", delete_flag ? "Untagging" : "Tagging",
               update_dir);
    return (R_PROCESS);
}
```

A.53 run.c

```
/* run.c — routines for executing subprocesses.

   This file is part of GNU CVS.

   GNU CVS is free software; you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published by the
   Free Software Foundation; either version 2, or (at your option) any
   later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include "cvs.h"

#ifndef HAVE_UNISTD_H
extern int execvp PROTO((char *file, char **argv));
#endif

static void run_add_arg PROTO((const char *s));

extern char *strtok ();

/*
 * To exec a program under CVS, first call run_setup() to setup initial
 * arguments.  The argument to run_setup will be parsed into whitespace
 * separated words and added to the global run_argv list.
 *
 * Then, optionally call run_arg() for each additional argument that you'd like
 * to pass to the executed program.
 *
 * Finally, call run_exec() to execute the program with the specified arguments.
 * The execvp() syscall will be used, so that the PATH is searched correctly.
 * File redirections can be performed in the call to run_exec().
 */
static char **run_argv;
static int run_argc;
static int run_argc_allocated;

/* VARARGS */
void
run_setup (prog)
    const char *prog;
{
    char *cp;
    int i;
    char *run_prog;

    /* clean out any malloc'ed values from run_argv */
    for (i = 0; i < run_argc; i++)
    {
        if (run_argv[i])
        {
            free (run_argv[i]);
            run_argv[i] = (char *) 0;
        }
    }
    run_argc = 0;

    run_prog = xstrdup (prog);

    /* put each word into run_argv, allocating it as we go */
    for (cp = strtok (run_prog, " \t"); cp; cp = strtok ((char *) NULL, " \t"))
        run_add_arg (cp);
    free (run_prog);
}

void
run_arg (s)
    const char *s;
{
    run_add_arg (s);
}

static void
run_add_arg (s)
    const char *s;
{
    /* allocate more argv entries if we've run out */
    if (run_argc >= run_argc_allocated)
    {
        run_argc_allocated += 50;
        run_argv = (char **) xrealloc ((char *) run_argv,
                                       run_argc_allocated * sizeof (char **));
    }

    if (s)
```

```
        run_argv[run_argc++] = xstrdup (s);
    else
        run_argv[run_argc] = (char *) 0; /* not post-incremented on purpose! */
}

int
run_exec (stin, stout, sterr, flags)
    const char *stin;
    const char *stout;
    const char *sterr;
    int flags;
{
    int shin, shout, sherr;
    int mode_out, mode_err;
    int status;
    int rc = -1;
    int rerrno = 0;
    int pid, w;

#ifdef POSIX_SIGNALS
    sigset_t sigset_mask, sigset_omask;
    struct sigaction act, iact, qact;

#else
#ifdef BSD_SIGNALS
    int mask;
    struct sigvec vec, ivec, qvec;

#else
    RETSIGTYPE (*istat) (), (*qstat) ();
#endif
#endif

    if (trace)
    {
#ifdef SERVER_SUPPORT
        cvs_outerr (server_active ? "S" : " ", 1);
#endif
        cvs_outerr ("-> system(", 0);
        run_print (stderr);
        cvs_outerr (")\n", 0);
    }
    if (noexec && (flags & RUN_REALLY) == 0)
        return (0);

    /* make sure that we are null terminated, since we didn't calloc */
    run_add_arg ((char *) 0);

    /* setup default file descriptor numbers */
    shin = 0;
    shout = 1;
    sherr = 2;

    /* set the file modes for stdout and stderr */
    mode_out = mode_err = O_WRONLY | O_CREAT;
    mode_out |= ((flags & RUN_STDOUT_APPEND) ? O_APPEND : O_TRUNC);
    mode_err |= ((flags & RUN_STDERR_APPEND) ? O_APPEND : O_TRUNC);

    if (stin && (shin = open (stin, O_RDONLY)) == -1)
    {
        rerrno = errno;
        error (0, errno, "cannot open %s for reading (prog %s)",
               stin, run_argv[0]);
        goto out0;
    }
    if (stout && (shout = open (stout, mode_out, 0666)) == -1)
    {
        rerrno = errno;
        error (0, errno, "cannot open %s for writing (prog %s)",
               stout, run_argv[0]);
        goto out1;
    }
    if (sterr && (flags & RUN_COMBINED) == 0)
    {
        if ((sherr = open (sterr, mode_err, 0666)) == -1)
        {
            rerrno = errno;
            error (0, errno, "cannot open %s for writing (prog %s)",
                   sterr, run_argv[0]);
            goto out2;
        }
    }

    /* Make sure we don't flush this twice, once in the subprocess. */
    fflush (stdout);
    fflush (stderr);

    /* The output files, if any, are now created.  Do the fork and dups.

       We use vfork not so much for the sake of unices without
```

```
        copy-on-write (such systems are rare these days), but for the
        sake of systems without an MMU, which therefore can't do
        copy-on-write (e.g. Amiga).  The other solution is spawn (see
        windows-NT/run.c).  */

    #ifdef HAVE_VFORK
        pid = vfork ();
    #else
        pid = fork ();
    #endif
        if (pid == 0)
        {
            if (shin != 0)
            {
                (void) dup2 (shin, 0);
                (void) close (shin);
            }
            if (shout != 1)
            {
                (void) dup2 (shout, 1);
                (void) close (shout);
            }
            if (flags & RUN_COMBINED)
                (void) dup2 (1, 2);
            else if (sherr != 2)
            {
                (void) dup2 (sherr, 2);
                (void) close (sherr);
            }

            /* dup'ing is done.  try to run it now */
            (void) execvp (run_argv[0], run_argv);
            error (0, errno, "cannot exec %s", run_argv[0]);
            _exit (127);
        }
        else if (pid == -1)
        {
            rerrno = errno;
            goto out;
        }

        /* the parent.  Ignore some signals for now */
    #ifdef POSIX_SIGNALS
        if (flags & RUN_SIGIGNORE)
        {
            act.sa_handler = SIG_IGN;
            (void) sigemptyset (&act.sa_mask);
            act.sa_flags = 0;
            (void) sigaction (SIGINT, &act, &iact);
            (void) sigaction (SIGQUIT, &act, &qact);
        }
        else
        {
            (void) sigemptyset (&sigset_mask);
            (void) sigaddset (&sigset_mask, SIGINT);
            (void) sigaddset (&sigset_mask, SIGQUIT);
            (void) sigprocmask (SIG_SETMASK, &sigset_mask, &sigset_omask);
        }
    #else
    #ifdef BSD_SIGNALS
        if (flags & RUN_SIGIGNORE)
        {
            memset ((char *) &vec, 0, sizeof (vec));
            vec.sv_handler = SIG_IGN;
            (void) sigvec (SIGINT, &vec, &ivec);
            (void) sigvec (SIGQUIT, &vec, &qvec);
        }
        else
            mask = sigblock (sigmask (SIGINT) | sigmask (SIGQUIT));
    #else
        istat = signal (SIGINT, SIG_IGN);
        qstat = signal (SIGQUIT, SIG_IGN);
    #endif
    #endif

        /* wait for our process to die and munge return status */
    #ifdef POSIX_SIGNALS
        while ((w = waitpid (pid, &status, 0)) == -1 && errno == EINTR)
            ;
    #else
        while ((w = wait (&status)) != pid)
        {
            if (w == -1 && errno != EINTR)
                break;
        }
    #endif

        if (w == -1)
        {
            rc = -1;
```

```
        rerrno = errno;
    }
#ifndef VMS /* status is return status */
    else if (WIFEXITED (status))
        rc = WEXITSTATUS (status);
    else if (WIFSIGNALED (status))
    {
        if (WTERMSIG (status) == SIGPIPE)
            error (1, 0, "broken pipe");
        rc = 2;
    }
    else
        rc = 1;
#else /* VMS */
    rc = WEXITSTATUS (status);
#endif /* VMS */

    /* restore the signals */
#ifdef POSIX_SIGNALS
    if (flags & RUN_SIGIGNORE)
    {
        (void) sigaction (SIGINT, &iact, (struct sigaction *) NULL);
        (void) sigaction (SIGQUIT, &qact, (struct sigaction *) NULL);
    }
    else
        (void) sigprocmask (SIG_SETMASK, &sigset_omask, (sigset_t *) NULL);
#else
#ifdef BSD_SIGNALS
    if (flags & RUN_SIGIGNORE)
    {
        (void) sigvec (SIGINT, &ivec, (struct sigvec *) NULL);
        (void) sigvec (SIGQUIT, &qvec, (struct sigvec *) NULL);
    }
    else
        (void) sigsetmask (mask);
#else
    (void) signal (SIGINT, istat);
    (void) signal (SIGQUIT, qstat);
#endif
#endif

    /* cleanup the open file descriptors */
  out:
    if (sterr)
        (void) close (sherr);
  out2:
    if (stout)
        (void) close (shout);
  out1:
    if (stin)
        (void) close (shin);

  out0:
    if (rerrno)
        errno = rerrno;
    return (rc);
}

void
run_print (fp)
    FILE *fp;
{
    int i;
    void (*outfn) PROTO ((const char *, size_t));

    if (fp == stderr)
        outfn = cvs_outerr;
    else if (fp == stdout)
        outfn = cvs_output;
    else
    {
        error (1, 0, "internal error: bad argument to run_print");
        /* Solely to placate gcc -Wall.
           FIXME: it'd be better to use a function named 'fatal' that
           is known never to return.  Then kludges wouldn't be necessary.  */
        outfn = NULL;
    }

    for (i = 0; i < run_argc; i++)
    {
        (*outfn) ("'", 1);
        (*outfn) (run_argv[i], 0);
        (*outfn) ("'", 1);
        if (i != run_argc - 1)
            (*outfn) (" ", 1);
    }
}

/* Return value is NULL for error, or if noexec was set.  If there was an
   error, return NULL and I'm not sure whether errno was set (the Red Hat
```

```
   Linux 4.1 popen manpage was kind of vague but discouraging; and the noexec
   case complicates this even aside from popen behavior).  */

FILE *
run_popen (cmd, mode)
    const char *cmd;
    const char *mode;
{
    if (trace)
#ifdef SERVER_SUPPORT
        (void) fprintf (stderr, "%c-> run_popen(%s,%s)\n",
                        (server_active) ? 'S' : ' ', cmd, mode);
#else
        (void) fprintf (stderr, "-> run_popen(%s,%s)\n", cmd, mode);
#endif
    if (noexec)
        return (NULL);

    return (popen (cmd, mode));
}

int
piped_child (command, tofdp, fromfdp)
     char **command;
     int *tofdp;
     int *fromfdp;
{
    int pid;
    int to_child_pipe[2];
    int from_child_pipe[2];

    if (pipe (to_child_pipe) < 0)
        error (1, errno, "cannot create pipe");
    if (pipe (from_child_pipe) < 0)
        error (1, errno, "cannot create pipe");

#ifdef USE_SETMODE_BINARY
    setmode (to_child_pipe[0], O_BINARY);
    setmode (to_child_pipe[1], O_BINARY);
    setmode (from_child_pipe[0], O_BINARY);
    setmode (from_child_pipe[1], O_BINARY);
#endif

#ifdef HAVE_VFORK
    pid = vfork ();
#else
    pid = fork ();
#endif
    if (pid < 0)
        error (1, errno, "cannot fork");
    if (pid == 0)
    {
        if (dup2 (to_child_pipe[0], STDIN_FILENO) < 0)
            error (1, errno, "cannot dup2");
        if (close (to_child_pipe[1]) < 0)
            error (1, errno, "cannot close");
        if (close (from_child_pipe[0]) < 0)
            error (1, errno, "cannot close");
        if (dup2 (from_child_pipe[1], STDOUT_FILENO) < 0)
            error (1, errno, "cannot dup2");

        execvp (command[0], command);
        error (1, errno, "cannot exec");
    }
    if (close (to_child_pipe[0]) < 0)
        error (1, errno, "cannot close");
    if (close (from_child_pipe[1]) < 0)
        error (1, errno, "cannot close");

    *tofdp = to_child_pipe[1];
    *fromfdp = from_child_pipe[0];
    return pid;
}


void
close_on_exec (fd)
     int fd;
{
#if defined (FD_CLOEXEC) && defined (F_SETFD)
  if (fcntl (fd, F_SETFD, 1))
    error (1, errno, "can't set close-on-exec flag on %d", fd);
#endif
}

/*
 * dir = 0 : main proc writes to new proc, which writes to oldfd
 * dir = 1 : main proc reads from new proc, which reads from oldfd
 *
 * Returns: a file descriptor.  On failure (i.e., the exec fails),
```

```
 * then filter_stream_through_program() complains and dies.
 */

int
filter_stream_through_program (oldfd, dir, prog, pidp)
     int oldfd, dir;
     char **prog;
     pid_t *pidp;
{
    int p[2], newfd;
    pid_t newpid;

    if (pipe (p))
        error (1, errno, "cannot create pipe");
#ifdef USE_SETMODE_BINARY
    setmode (p[0], O_BINARY);
    setmode (p[1], O_BINARY);
#endif

#ifdef HAVE_VFORK
    newpid = vfork ();
#else
    newpid = fork ();
#endif
    if (pidp)
        *pidp = newpid;
    switch (newpid)
    {
      case -1:
        error (1, errno, "cannot fork");
      case 0:
        /* child */
        if (dir)
        {
            /* write to new pipe */
            close (p[0]);
            dup2 (oldfd, 0);
            dup2 (p[1], 1);
        }
        else
        {
            /* read from new pipe */
            close (p[1]);
            dup2 (p[0], 0);
            dup2 (oldfd, 1);
        }
        /* Should I be blocking some signals here?  */
        execvp (prog[0], prog);
        error (1, errno, "couldn't exec %s", prog[0]);
      default:
        /* parent */
        close (oldfd);
        if (dir)
        {
            /* read from new pipe */
            close (p[1]);
            newfd = p[0];
        }
        else
        {
            /* write to new pipe */
            close (p[0]);
            newfd = p[1];
        }
        close_on_exec (newfd);
        return newfd;
    }
}
```

A.54 scramble.c

```
/*
 * Trivially encode strings to protect them from innocent eyes (i.e.,
 * inadvertent password compromises, like a network administrator
 * who's watching packets for legitimate reasons and accidentally sees
 * the password protocol go by).
 *
 * This is NOT secure encryption.
 *
 * It would be tempting to encode the password according to username
 * and repository, so that the same password would encode to a
 * different string when used with different usernames and/or
 * repositories.  However, then users would not be able to cut and
 * paste passwords around.  They're not supposed to anyway, but we all
 * know they will, and there's no reason to make it harder for them if
 * we're not trying to provide real security anyway.
 */

/* Set this to test as a standalone program. */
/* #define DIAGNOSTIC */

#ifndef DIAGNOSTIC
#include "cvs.h"
#else /* ! DIAGNOSTIC */
/* cvs.h won't define this for us */
#define AUTH_CLIENT_SUPPORT
#define xmalloc malloc
/* Use "gcc -fwritable-strings". */
#include <stdio.h>
#include <stdio.h>
#include <string.h>
#endif /* ! DIAGNOSTIC */

#if defined(AUTH_CLIENT_SUPPORT) || defined(AUTH_SERVER_SUPPORT)

/* Map characters to each other randomly and symmetrically, A <-> B.
 *
 * We divide the ASCII character set into 3 domains: control chars (0
 * thru 31), printing chars (32 through 126), and "meta"-chars (127
 * through 255).  The control chars map _to_ themselves, the printing
 * chars map _among_ themselves, and the meta chars map _among_
 * themselves.  Why is this thus?
 *
 * No character in any of these domains maps to a character in another
 * domain, because I'm not sure what characters are legal in
 * passwords, or what tools people are likely to use to cut and paste
 * them.  It seems prudent not to introduce control or meta chars,
 * unless the user introduced them first.  And having the control
 * chars all map to themselves insures that newline and
 * carriage-return are safely handled.
 */

static unsigned char
shifts[] = {
    0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
   16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
  114,120, 53, 79, 96,109, 72,108, 70, 64, 76, 67,116, 74, 68, 87,
  111, 52, 75,119, 49, 34, 82, 81, 95, 65,112, 86,118,110,122,105,
   41, 57, 83, 43, 46,102, 40, 89, 38,103, 45, 50, 42,123, 91, 35,
  125, 55, 54, 66,124,126, 59, 47, 92, 71,115, 78, 88,107,106, 56,
   36,121,117,104,101,100, 69, 73, 99, 63, 94, 93, 39, 37, 61, 48,
   58,113, 32, 90, 44, 98, 60, 51, 33, 97, 62, 77, 84, 80, 85,223,
  225,216,187,166,229,189,222,188,141,249,148,200,184,136,248,190,
  199,170,181,204,138,232,218,183,255,234,220,247,213,203,226,193,
  174,172,228,252,217,201,131,230,197,211,145,238,161,179,160,212,
  207,221,254,173,202,146,224,151,140,196,205,130,135,133,143,246,
  192,159,244,239,185,168,215,144,139,165,180,157,147,186,214,176,
  227,231,219,169,175,156,206,198,129,164,150,210,154,177,134,127,
  182,128,158,208,162,132,167,209,149,241,153,251,237,236,171,195,
  243,233,253,240,194,250,191,155,142,137,245,235,163,242,178,152 };

/* SCRAMBLE and DESCRAMBLE work like this:
 *
 * scramble(STR) returns SCRM, a scrambled copy of STR.  SCRM[0] is a
 * single letter indicating the scrambling method.  As of this
 * writing, the only legal method is 'A', but check the code for more
 * up-to-date information.  The copy will have been allocated with
 * malloc().
 *
 * descramble(SCRM) returns STR, again in its own malloc'd space.
 * descramble() uses SCRM[0] to determine which method of unscrambling
 * to use.  If it does not recognize the method, it dies with error.
 */

/* Return a malloc'd, scrambled version of STR. */
char *
scramble (str)
    char *str;
```

```
{
    int i;
    char *s;

    /* +2 to hold the 'A' prefix that indicates which version of
       scrambling this is (the first, obviously, since we only do one
       kind of scrambling so far), and then the '\0' of course. */
    s = (char *) xmalloc (strlen (str) + 2);

    /* Scramble (TM) version prefix. */
    s[0] = 'A';
    strcpy (s + 1, str);

    for (i = 1; s[i]; i++)
        s[i] = shifts[(unsigned char)(s[i])];

    return s;
}

/* Decode the string in place. */
char *
descramble (str)
    char *str;
{
    char *s;
    int i;

    /* For now we can only handle one kind of scrambling.  In the future
       there may be other kinds, and this 'if' will become a 'switch'.  */
    if (str[0] != 'A')
#ifndef DIAGNOSTIC
        error (1, 0, "descramble: unknown scrambling method");
#else  /* DIAGNOSTIC */
    {
        fprintf (stderr, "descramble: unknown scrambling method\n", str);
        fflush (stderr);
        exit (EXIT_FAILURE);
    }
#endif  /* DIAGNOSTIC */

    /* Method 'A' is symmetrical, so scramble again to decrypt. */
    s = scramble (str + 1);

    /* Shift the whole string one char to the left, pushing the unwanted
       'A' off the left end.  Safe, because s is null-terminated. */
    for (i = 0; s[i]; i++)
        s[i] = s[i + 1];

    return s;
}

#endif /* (AUTH_CLIENT_SUPPORT || AUTH_SERVER_SUPPORT) from top of file */

#ifdef DIAGNOSTIC
int
main ()
{
    int i;
    char *e, *m, biggie[256];

    char *cleartexts[5];
    cleartexts[0] = "first";
    cleartexts[1] = "the second";
    cleartexts[2] = "this is the third";
    cleartexts[3] = "$#% !!\\3";
    cleartexts[4] = biggie;

    /* Set up the most important test string: */
    /* Can't have a real ASCII zero in the string, because we want to
       use printf, so we substitute the character zero. */
    biggie[0] = '0';
    /* The rest of the string gets straight ascending ASCII. */
    for (i = 1; i < 256; i++)
        biggie[i] = i;

    /* Test all the strings. */
    for (i = 0; i < 5; i++)
    {
        printf ("clear%d: %s\n", i, cleartexts[i]);
        e = scramble (cleartexts[i]);
        printf ("scram%d: %s\n", i, e);
        m = descramble (e);
        free (e);
        printf ("clear%d: %s\n\n", i, m);
        free (m);
    }

    fflush (stdout);
    return 0;
}
```

```
#endif /* DIAGNOSTIC */

/*
 * ;;; The Emacs Lisp that did the dirty work ;;;
 * (progn
 *
 *   ;; Helper func.
 *   (defun random-elt (lst)
 *     (let* ((len (length lst))
 *            (rnd (random len)))
 *       (nth rnd lst)))
 *
 *   ;; A list of all characters under 127, each appearing once.
 *   (setq non-meta-chars
 *         (let ((i 0)
 *               (l nil))
 *           (while (< i 127)
 *             (setq l (cons i l)
 *                   i (1+ i)))
 *           l))
 *
 *   ;; A list of all characters 127 and above, each appearing once.
 *   (setq meta-chars
 *         (let ((i 127)
 *               (l nil))
 *           (while (< i 256)
 *             (setq l (cons i l)
 *                   i (1+ i)))
 *           l))
 *
 *   ;; A vector that will hold the chars in a random order.
 *   (setq scrambled-chars (make-vector 256 0))
 *
 *   ;; These characters should map to themselves.
 *   (let ((i 0))
 *     (while (< i 32)
 *       (aset scrambled-chars i i)
 *       (setq non-meta-chars (delete i non-meta-chars)
 *             i (1+ i))))
 *
 *   ;; Assign random (but unique) values, within the non-meta chars.
 *   (let ((i 32))
 *     (while (< i 127)
 *       (let ((ch (random-elt non-meta-chars)))
 *         (if (= 0 (aref scrambled-chars i))
 *             (progn
 *               (aset scrambled-chars i ch)
 *               (aset scrambled-chars ch i)
 *               (setq non-meta-chars (delete ch non-meta-chars)
 *                     non-meta-chars (delete i non-meta-chars))))
 *         (setq i (1+ i)))))
 *
 *   ;; Assign random (but unique) values, within the non-meta chars.
 *   (let ((i 127))
 *     (while (< i 256)
 *       (let ((ch (random-elt meta-chars)))
 *         (if (= 0 (aref scrambled-chars i))
 *             (progn
 *               (aset scrambled-chars i ch)
 *               (aset scrambled-chars ch i)
 *               (setq meta-chars (delete ch meta-chars)
 *                     meta-chars (delete i meta-chars))))
 *         (setq i (1+ i)))))
 *
 *   ;; Now use the 'scrambled-chars' vector to get your C array.
 *   )
 */
```

A.55 server.c

```
/* This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include <assert.h>
#include "cvs.h"
#include "watch.h"
#include "edit.h"
#include "fileattr.h"
#include "getline.h"
#include "buffer.h"

#ifdef SERVER_SUPPORT

#ifdef HAVE_WINSOCK_H
#include <winsock.h>
#endif

#if defined (AUTH_SERVER_SUPPORT) || defined (HAVE_KERBEROS) || defined (HAVE_GSSAPI)
#include <sys/socket.h>
#endif

#ifdef HAVE_KERBEROS
# include <netinet/in.h>
# include <krb.h>
# ifndef HAVE_KRB_GET_ERR_TEXT
#   define krb_get_err_text(status) krb_err_txt[status]
# endif

/* Information we need if we are going to use Kerberos encryption.  */
static C_Block kblock;
static Key_schedule sched;

#endif

#ifdef HAVE_GSSAPI

#include <netdb.h>

#ifdef HAVE_GSSAPI_H
#include <gssapi.h>
#endif
#ifdef HAVE_GSSAPI_GSSAPI_H
#include <gssapi/gssapi.h>
#endif
#ifdef HAVE_GSSAPI_GSSAPI_GENERIC_H
#include <gssapi/gssapi_generic.h>
#endif

#ifndef HAVE_GSS_C_NT_HOSTBASED_SERVICE
#define GSS_C_NT_HOSTBASED_SERVICE gss_nt_service_name
#endif

/* We use Kerberos 5 routines to map the GSSAPI credential to a user
   name.  */
#include <krb5.h>

/* We need this to wrap data.  */
static gss_ctx_id_t gcontext;

static int gserver_authenticate_connection PROTO((int));
static int kserver_authenticate_connection PROTO((int));
static int pserver_authenticate_connection PROTO((int));

/* Whether we are already wrapping GSSAPI communication.  */
static int cvs_gssapi_wrapping;

# ifdef ENCRYPTION
/* Whether to encrypt GSSAPI communication.  We use a global variable
   like this because we use the same buffer type (gssapi_wrap) to
   handle both authentication and encryption, and we don't want
   multiple instances of that buffer in the communication stream.  */
int cvs_gssapi_encrypt;
# endif

#endif

/* for select */
#include <sys/types.h>
#ifdef HAVE_SYS_BSDTYPES_H
#include <sys/bsdtypes.h>
#endif
```

```
#if TIME_WITH_SYS_TIME
# include <sys/time.h>
# include <time.h>
#else
# if HAVE_SYS_TIME_H
#  include <sys/time.h>
# else
#  include <time.h>
# endif
#endif

#if HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#ifndef O_NONBLOCK
#define O_NONBLOCK O_NDELAY
#endif

/* EWOULDBLOCK is not defined by POSIX, but some BSD systems will
   return it, rather than EAGAIN, for nonblocking writes.  */
#ifdef EWOULDBLOCK
#define blocking_error(err) ((err) == EWOULDBLOCK || (err) == EAGAIN)
#else
#define blocking_error(err) ((err) == EAGAIN)
#endif

#ifdef AUTH_SERVER_SUPPORT
#ifdef HAVE_GETSPNAM
#include <shadow.h>
#endif
#endif /* AUTH_SERVER_SUPPORT */

/* For initgroups().  */
#if HAVE_INITGROUPS
#include <grp.h>
#endif /* HAVE_INITGROUPS */

#ifdef AUTH_SERVER_SUPPORT

/* The cvs username sent by the client, which might or might not be
   the same as the system username the server eventually switches to
   run as.  CVS_Username gets set iff password authentication is
   successful. */
char *CVS_Username = NULL;

/* Used to check that same repos is transmitted in pserver auth and in
   later CVS protocol.  Exported because root.c also uses. */
char *Pserver_Repos = NULL;

/* Should we check for system usernames/passwords?  Can be changed by
   CVSROOT/config.  */
int system_auth = 1;

#endif /* AUTH_SERVER_SUPPORT */


/* While processing requests, this buffer accumulates data to be sent to
   the client, and then once we are in do_cvs_command, we use it
   for all the data to be sent.  */
static struct buffer *buf_to_net;

/* This buffer is used to read input from the client.  */
static struct buffer *buf_from_net;

/*
 * This is where we stash stuff we are going to use.  Format string
 * which expects a single directory within it, starting with a slash.
 */
static char *server_temp_dir;

/* This is the original value of server_temp_dir, before any possible
   changes inserted by serve_max_dotdot.  */
static char *orig_server_temp_dir;

/* Nonzero if we should keep the temp directory around after we exit.  */
static int dont_delete_temp;

static void server_write_entries PROTO((void));

/* All server communication goes through buffer structures.  Most of
   the buffers are built on top of a file descriptor.  This structure
   is used as the closure field in a buffer.  */

struct fd_buffer
{
    /* The file descriptor.  */
    int fd;
    /* Nonzero if the file descriptor is in blocking mode.  */
```

```
    int blocking;
};

static struct buffer *fd_buffer_initialize
  PROTO ((int, int, void (*) (struct buffer *)));
static int fd_buffer_input PROTO((void *, char *, int, int, int *));
static int fd_buffer_output PROTO((void *, const char *, int, int *));
static int fd_buffer_flush PROTO((void *));
static int fd_buffer_block PROTO((void *, int));

/* Initialize a buffer built on a file descriptor.  FD is the file
   descriptor.  INPUT is nonzero if this is for input, zero if this is
   for output.  MEMORY is the function to call when a memory error
   occurs.  */

static struct buffer *
fd_buffer_initialize (fd, input, memory)
     int fd;
     int input;
     void (*memory) PROTO((struct buffer *));
{
    struct fd_buffer *n;

    n = (struct fd_buffer *) xmalloc (sizeof *n);
    n->fd = fd;
    n->blocking = 1;
    return buf_initialize (input ? fd_buffer_input : NULL,
                           input ? NULL : fd_buffer_output,
                           input ? NULL : fd_buffer_flush,
                           fd_buffer_block,
                           (int (*) PROTO((void *))) NULL,
                           memory,
                           n);
}

/* The buffer input function for a buffer built on a file descriptor.  */

static int
fd_buffer_input (closure, data, need, size, got)
     void *closure;
     char *data;
     int need;
     int size;
     int *got;
{
    struct fd_buffer *fd = (struct fd_buffer *) closure;
    int nbytes;

    if (! fd->blocking)
        nbytes = read (fd->fd, data, size);
    else
    {
        /* This case is not efficient.  Fortunately, I don't think it
           ever actually happens.  */
        nbytes = read (fd->fd, data, need == 0 ? 1 : need);
    }

    if (nbytes > 0)
    {
        *got = nbytes;
        return 0;
    }

    *got = 0;

    if (nbytes == 0)
    {
        /* End of file.  This assumes that we are using POSIX or BSD
           style nonblocking I/O.  On System V we will get a zero
           return if there is no data, even when not at EOF.  */
        return -1;
    }

    /* Some error occurred.  */

    if (blocking_error (errno))
    {
        /* Everything's fine, we just didn't get any data.  */
        return 0;
    }

    return errno;
}

/* The buffer output function for a buffer built on a file descriptor.  */

static int
fd_buffer_output (closure, data, have, wrote)
     void *closure;
     const char *data;
```

```
     int have;
270  int *wrote;
{
  struct fd_buffer *fd = (struct fd_buffer *) closure;

  *wrote = 0;

  while (have > 0)
  {
    int nbytes;

280 nbytes = write (fd->fd, data, have);

    if (nbytes <= 0)
    {
      if (! fd->blocking
          && (nbytes == 0 || blocking_error (errno)))
      {
        /* A nonblocking write failed to write any data.  Just
           return.  */
        return 0;
290   }

      /* Some sort of error occurred.  */

      if (nbytes == 0)
        return EIO;

      return errno;
    }

300 *wrote += nbytes;
    data += nbytes;
    have -= nbytes;
  }

  return 0;
}

/* The buffer flush function for a buffer built on a file descriptor.  */

310 /*ARGSUSED*/
static int
fd_buffer_flush (closure)
     void *closure;
{
  /* Nothing to do.  File descriptors are always flushed.  */
  return 0;
}

/* The buffer block function for a buffer built on a file descriptor.  */
320
static int
fd_buffer_block (closure, block)
     void *closure;
     int block;
{
  struct fd_buffer *fd = (struct fd_buffer *) closure;
  int flags;

  flags = fcntl (fd->fd, F_GETFL, 0);
330 if (flags < 0)
    return errno;

  if (block)
    flags &= ~O_NONBLOCK;
  else
    flags |= O_NONBLOCK;

  if (fcntl (fd->fd, F_SETFL, flags) < 0)
    return errno;
340
  fd->blocking = block;

  return 0;
}

/*
 * Make directory DIR, including all intermediate directories if necessary.
 * Returns 0 for success or errno code.
 */
350 static int mkdir_p PROTO((char *));

static int
mkdir_p (dir)
     char *dir;
{
  char *p;
  char *q = malloc (strlen (dir) + 1);
  int retval;
```

```
360     if (q == NULL)
            return ENOMEM;

        retval = 0;

        /*
         * Skip over leading slash if present.  We won't bother to try to
         * make '/'.
         */
        p = dir + 1;
370     while (1)
        {
            while (*p != '/' && *p != '\0')
                ++p;
            if (*p == '/')
            {
                strncpy (q, dir, p - dir);
                q[p - dir] = '\0';
                if (q[p - dir - 1] != '/'  &&  CVS_MKDIR (q, 0777) < 0)
                {
380                 int saved_errno = errno;

                    if (saved_errno != EEXIST
                        && ((saved_errno != EACCES && saved_errno != EROFS)
                            || !isdir (q)))
                    {
                        retval = saved_errno;
                        goto done;
                    }
                }
390             ++p;
            }
            else
            {
                if (CVS_MKDIR (dir, 0777) < 0)
                    retval = errno;
                goto done;
            }
        }
      done:
400     free (q);
        return retval;
    }

    /*
     * Print the error response for error code STATUS.  The caller is
     * reponsible for making sure we get back to the command loop without
     * any further output occuring.
     * Must be called only in contexts where it is OK to send output.
     */
410 static void
    print_error (status)
        int status;
    {
        char *msg;
        buf_output0 (buf_to_net, "error  ");
        msg = strerror (status);
        if (msg)
            buf_output0 (buf_to_net, msg);
        buf_append_char (buf_to_net, '\n');
420
        buf_flush (buf_to_net, 0);
    }

    static int pending_error;
    /*
     * Malloc'd text for pending error.  Each line must start with "E ".  The
     * last line should not end with a newline.
     */
    static char *pending_error_text;
430
    /* If an error is pending, print it and return 1.  If not, return 0.
       Must be called only in contexts where it is OK to send output.  */
    static int
    print_pending_error ()
    {
        if (pending_error_text)
        {
            buf_output0 (buf_to_net, pending_error_text);
            buf_append_char (buf_to_net, '\n');
440         if (pending_error)
                print_error (pending_error);
            else
                buf_output0 (buf_to_net, "error  \n");

            buf_flush (buf_to_net, 0);

            pending_error = 0;
            free (pending_error_text);
```

```
        pending_error_text = NULL;
        return 1;
    }
    else if (pending_error)
    {
        print_error (pending_error);
        pending_error = 0;
        return 1;
    }
    else
        return 0;
}

/* Is an error pending?  */
#define error_pending() (pending_error || pending_error_text)

static int alloc_pending PROTO ((size_t size));

/* Allocate SIZE bytes for pending_error_text and return nonzero
   if we could do it.  */
static int
alloc_pending (size)
    size_t size;
{
    if (error_pending ())
        /* Probably alloc_pending callers will have already checked for
           this case.  But we might as well handle it if they don't, I
           guess.  */
        return 0;
    pending_error_text = malloc (size);
    if (pending_error_text == NULL)
    {
        pending_error = ENOMEM;
        return 0;
    }
    return 1;
}

static void serve_is_modified PROTO ((char *));

static int supported_response PROTO ((char *));

static int
supported_response (name)
     char *name;
{
    struct response *rs;

    for (rs = responses; rs->name != NULL; ++rs)
        if (strcmp (rs->name, name) == 0)
            return rs->status == rs_supported;
    error (1, 0, "internal error: testing support for unknown response?");
    /* NOTREACHED */
    return 0;
}

static void
serve_valid_responses (arg)
     char *arg;
{
    char *p = arg;
    char *q;
    struct response *rs;
    do
    {
        q = strchr (p, ' ');
        if (q != NULL)
            *q++ = '\0';
        for (rs = responses; rs->name != NULL; ++rs)
        {
            if (strcmp (rs->name, p) == 0)
                break;
        }
        if (rs->name == NULL)
            /*
             * It is a response we have never heard of (and thus never
             * will want to use).  So don't worry about it.
             */
            ;
        else
            rs->status = rs_supported;
        p = q;
    } while (q != NULL);
    for (rs = responses; rs->name != NULL; ++rs)
    {
        if (rs->status == rs_essential)
        {
            buf_output0 (buf_to_net, "E response `");
            buf_output0 (buf_to_net, rs->name);
            buf_output0 (buf_to_net, "' not supported by client\nerror  \n");
```

```
            /* FIXME: This call to buf_flush could conceivably
               cause deadlock, as noted in server_cleanup.  */
            buf_flush (buf_to_net, 1);

            /* I'm doing this manually rather than via error_exit ()
               because I'm not sure whether we want to call server_cleanup.
               Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
            /* Hook for OS-specific behavior, for example socket subsystems on
               NT and OS2 or dealing with windows and arguments on Mac.  */
            SYSTEM_CLEANUP ();
#endif

            exit (EXIT_FAILURE);
        }
        else if (rs->status == rs_optional)
            rs->status = rs_not_supported;
    }
}

static void
serve_root (arg)
    char *arg;
{
    char *env;
    char *path;
    int save_errno;

    if (error_pending()) return;

    if (!isabsolute (arg))
    {
        if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text,
                     "E Root %s must be an absolute pathname", arg);
        return;
    }

    /* Sending "Root" twice is illegal.  It would also be nice to
       check for the other case, in which there is no Root request
       prior to a request which requires one.

       The other way to handle a duplicate Root requests would be as a
       request to clear out all state and start over as if it was a
       new connection.  Doing this would cause interoperability
       headaches, so it should be a different request, if there is
       any reason why such a feature is needed.  */
    if (CVSroot_directory != NULL)
    {
        if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text,
                     "E Protocol error: Duplicate Root request, for %s", arg);
        return;
    }

    set_local_cvsroot (arg);

    /* For pserver, this will already have happened, and the call will do
       nothing.  But for rsh, we need to do it now.  */
    parse_config (CVSroot_directory);

    path = xmalloc (strlen (CVSroot_directory)
                    + sizeof (CVSROOTADM)
                    + sizeof (CVSROOTADM_HISTORY)
                    + 10);
    (void) sprintf (path, "%s/%s", CVSroot_directory, CVSROOTADM);
    if (!isaccessible (path, R_OK | X_OK))
    {
        save_errno = errno;
        pending_error_text = malloc (80 + strlen (path));
        if (pending_error_text != NULL)
            sprintf (pending_error_text, "E Cannot access %s", path);
        pending_error = save_errno;
    }
    (void) strcat (path, "/");
    (void) strcat (path, CVSROOTADM_HISTORY);
    if (isfile (path) && !isaccessible (path, R_OK | W_OK))
    {
        save_errno = errno;
        pending_error_text = malloc (80 + strlen (path));
        if (pending_error_text != NULL)
            sprintf (pending_error_text, "E \
Sorry, you don't have read/write access to the history file %s", path);
        pending_error = save_errno;
    }
    free (path);

#ifdef HAVE_PUTENV
```

```
    env = malloc (strlen (CVSROOT_ENV) + strlen (CVSroot_directory) + 1 + 1);
    if (env == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    (void) sprintf (env, "%s=%s", CVSROOT_ENV, CVSroot_directory);
    (void) putenv (env);
    /* do not free env, as putenv has control of it */
#endif
}

static int max_dotdot_limit = 0;

/* Is this pathname OK to recurse into when we are running as the server?
   If not, call error() with a fatal error.  */
void
server_pathname_check (path)
    char *path;
{
    /* An absolute pathname is almost surely a path on the *client* machine,
       and is unlikely to do us any good here.  It also is probably capable
       of being a security hole in the anonymous readonly case.  */
    if (isabsolute (path))
        /* Giving an error is actually kind of a cop-out, in the sense
           that it would be nice for "cvs co -d /foo/bar/baz" to work.
           A quick fix in the server would be requiring Max-dotdot of
           at least one if pathnames are absolute, and then putting
           /abs/foo/bar/baz in the temp dir beside the /d/d/d stuff.
           A cleaner fix in the server might be to decouple the
           pathnames we pass back to the client from pathnames in our
           temp directory (this would also probably remove the need
           for Max-dotdot).  A fix in the client would have the client
           turn it into "cd /foo/bar; cvs co -d baz" (more or less).
           This probably has some problems with pathnames which appear
           in messages.  */
        error (1, 0, "absolute pathname `%s' illegal for server", path);
    if (pathname_levels (path) > max_dotdot_limit)
    {
        /* Similar to the isabsolute case in security implications.  */
        error (0, 0, "protocol error: `%s' contains more leading ..", path);
        error (1, 0, "than the %d which Max-dotdot specified",
               max_dotdot_limit);
    }
}

/*
 * Add as many directories to the temp directory as the client tells us it
 * will use "..", so we never try to access something outside the temp
 * directory via "..".
 */
static void
serve_max_dotdot (arg)
    char *arg;
{
    int lim = atoi (arg);
    int i;
    char *p;

    if (lim < 0)
        return;
    p = malloc (strlen (server_temp_dir) + 2 * lim + 10);
    if (p == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (p, server_temp_dir);
    for (i = 0; i < lim; ++i)
        strcat (p, "/d");
    if (server_temp_dir != orig_server_temp_dir)
        free (server_temp_dir);
    server_temp_dir = p;
    max_dotdot_limit = lim;
}

static char *dir_name;

static void
dirswitch (dir, repos)
    char *dir;
    char *repos;
{
    int status;
    FILE *f;
    char *b;
    size_t dir_len;

    server_write_entries ();
```

```
    if (error_pending()) return;

    if (dir_name != NULL)
        free (dir_name);

    dir_len = strlen (dir);

    /* Check for a trailing '/'.  This is not ISDIRSEP because \ in the
       protocol is an ordinary character, not a directory separator (of
       course, it is perhaps unwise to use it in directory names, but that
       is another issue).  */
    if (dir_len > 0
        && dir[dir_len - 1] == '/')
    {
        if (alloc_pending (80 + dir_len))
            sprintf (pending_error_text,
                     "E protocol error: invalid directory syntax in %s", dir);
        return;
    }

    dir_name = malloc (strlen (server_temp_dir) + dir_len + 40);
    if (dir_name == NULL)
    {
        pending_error = ENOMEM;
        return;
    }

    strcpy (dir_name, server_temp_dir);
    strcat (dir_name, "/");
    strcat (dir_name, dir);

    status = mkdir_p (dir_name);
    if (status != 0
        && status != EEXIST)
    {
        pending_error = status;
        if (alloc_pending (80 + strlen (dir_name)))
            sprintf (pending_error_text, "E cannot mkdir %s", dir_name);
        return;
    }

    /* Note that this call to Subdir_Register will be a noop if the parent
       directory does not yet exist (for example, if the client sends
       "Directory foo" followed by "Directory .", then the subdirectory does
       not get registered, but if the client sends "Directory ." followed
       by "Directory foo", then the subdirectory does get registered.
       This seems pretty fishy, but maybe it is the way it needs to work.  */
    b = strrchr (dir_name, '/');
    *b = '\0';
    Subdir_Register ((List *) NULL, dir_name, b + 1);
    *b = '/';

    if ( CVS_CHDIR (dir_name) < 0)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (dir_name)))
            sprintf (pending_error_text, "E cannot change to %s", dir_name);
        return;
    }
    /*
     * This is pretty much like calling Create_Admin, but Create_Admin doesn't
     * report errors in the right way for us.
     */
    if (CVS_MKDIR (CVSADM, 0777) < 0)
    {
        if (errno == EEXIST)
            /* Don't create the files again.  */
            return;
        pending_error = errno;
        return;
    }
    f = CVS_FOPEN (CVSADM_REP, "w");
    if (f == NULL)
    {
        pending_error = errno;
        return;
    }
    if (fprintf (f, "%s", repos) < 0)
    {
        pending_error = errno;
        fclose (f);
        return;
    }
    /* Non-remote CVS handles a module representing the entire tree
       (e.g., an entry like ``world -a .'') by putting /. at the end
       of the Repository file, so we do the same.  */
    if (strcmp (dir, ".") == 0
        && CVSroot_directory != NULL
        && strcmp (CVSroot_directory, repos) == 0)
    {
```

```
        if (fprintf (f, "/.") < 0)
        {
            pending_error = errno;
            fclose (f);
            return;
        }
    }
    if (fprintf (f, "\n") < 0)
    {
        pending_error = errno;
        fclose (f);
        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        return;
    }
    /* We open in append mode because we don't want to clobber an
       existing Entries file.  */
    f = CVS_FOPEN (CVSADM_ENT, "a");
    if (f == NULL)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_ENT)))
            sprintf (pending_error_text, "E cannot open %s", CVSADM_ENT);
        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_ENT)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_ENT);
        return;
    }
}

static void
serve_repository (arg)
    char *arg;
{
    pending_error_text = malloc (80);
    if (pending_error_text == NULL)
        pending_error = ENOMEM;
    else
        strcpy (pending_error_text,
                "E Repository request is obsolete; aborted");
    return;
}

static void
serve_directory (arg)
    char *arg;
{
    int status;
    char *repos;

    status = buf_read_line (buf_from_net, &repos, (int *) NULL);
    if (status == 0)
    {
        /* I think isabsolute (repos) should always be true, and that
           any RELATIVE_REPOS stuff should only be in CVS/Repository
           files, not the protocol (for compatibility), but I'm putting
           in the in isabsolute check just in case.  */
        if (isabsolute (repos)
            && strncmp (CVSroot_directory,
                        repos,
                        strlen (CVSroot_directory)) != 0)
        {
            if (alloc_pending (strlen (CVSroot_directory)
                               + strlen (repos)
                               + 80))
                sprintf (pending_error_text, "\
E protocol error: directory '%s' not within root '%s'",
                         repos, CVSroot_directory);
            return;
        }

        dirswitch (arg, repos);
        free (repos);
    }
    else if (status == -2)
    {
        pending_error = ENOMEM;
    }
    else
    {
        pending_error_text = malloc (80 + strlen (arg));
        if (pending_error_text == NULL)
        {
```

```
            pending_error = ENOMEM;
        }
        else if (status == -1)
        {
            sprintf (pending_error_text,
                     "E end of file reading mode for %s", arg);
        }
        else
        {
            sprintf (pending_error_text,
                     "E error reading mode for %s", arg);
            pending_error = status;
        }
    }
}

static void
serve_static_directory (arg)
    char *arg;
{
    FILE *f;

    if (error_pending ()) return;

    f = CVS_FOPEN (CVSADM_ENTSTAT, "w+");
    if (f == NULL)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_ENTSTAT)))
            sprintf (pending_error_text, "E cannot open %s", CVSADM_ENTSTAT);
        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_ENTSTAT)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_ENTSTAT);
        return;
    }
}

static void
serve_sticky (arg)
    char *arg;
{
    FILE *f;

    if (error_pending ()) return;

    f = CVS_FOPEN (CVSADM_TAG, "w+");
    if (f == NULL)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_TAG)))
            sprintf (pending_error_text, "E cannot open %s", CVSADM_TAG);
        return;
    }
    if (fprintf (f, "%s\n", arg) < 0)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_TAG)))
            sprintf (pending_error_text, "E cannot write to %s", CVSADM_TAG);
        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_TAG)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_TAG);
        return;
    }
}

/*
 * Read SIZE bytes from buf_from_net, write them to FILE.
 *
 * Currently this isn't really used for receiving parts of a file --
 * the file is still sent over in one chunk.  But if/when we get
 * spiffy in-process gzip support working, perhaps the compressed
 * pieces could be sent over as they're ready, if the network is fast
 * enough.  Or something.
 */
static void
receive_partial_file (size, file)
     int size;
     int file;
{
    while (size > 0)
    {
        int status, nread;
```

```
        char *data;

        status = buf_read_data (buf_from_net, size, &data, &nread);
        if (status != 0)
        {
            if (status == -2)
                pending_error = ENOMEM;
            else
            {
                pending_error_text = malloc (80);
                if (pending_error_text == NULL)
                    pending_error = ENOMEM;
                else if (status == -1)
                {
                    sprintf (pending_error_text,
                             "E premature end of file from client");
                    pending_error = 0;
                }
                else
                {
                    sprintf (pending_error_text,
                             "E error reading from client");
                    pending_error = status;
                }
            }
            return;
        }

        size -= nread;

        while (nread > 0)
        {
            int nwrote;

            nwrote = write (file, data, nread);
            if (nwrote < 0)
            {
                pending_error_text = malloc (40);
                if (pending_error_text != NULL)
                    sprintf (pending_error_text, "E unable to write");
                pending_error = errno;

                /* Read and discard the file data.  */
                while (size > 0)
                {
                    int status, nread;
                    char *data;

                    status = buf_read_data (buf_from_net, size, &data, &nread);
                    if (status != 0)
                        return;
                    size -= nread;
                }

                return;
            }
            nread -= nwrote;
            data += nwrote;
        }
    }
}

/* Receive SIZE bytes, write to filename FILE.  */
static void
receive_file (size, file, gzipped)
     int size;
     char *file;
     int gzipped;
{
    int fd;
    char *arg = file;
    pid_t gzip_pid = 0;
    int gzip_status;

    /* Write the file.  */
    fd = CVS_OPEN (arg, O_WRONLY | O_CREAT | O_TRUNC, 0600);
    if (fd < 0)
    {
        pending_error_text = malloc (40 + strlen (arg));
        if (pending_error_text)
            sprintf (pending_error_text, "E cannot open %s", arg);
        pending_error = errno;
        return;
    }

    /*
     * FIXME: This doesn't do anything reasonable with gunzip's stderr, which
     * means that if gunzip writes to stderr, it will cause all manner of
     * protocol violations.
     */
```

```
    if (gzipped)
1080    fd = filter_through_gunzip (fd, 0, &gzip_pid);

    receive_partial_file (size, fd);

    if (pending_error_text)
    {
        char *p = realloc (pending_error_text,
                           strlen (pending_error_text) + strlen (arg) + 30);
        if (p)
        {
1090        pending_error_text = p;
            sprintf (p + strlen (p), ", file %s", arg);
        }
        /* else original string is supposed to be unchanged */
    }

    if (close (fd) < 0 && !error_pending ())
    {
        pending_error_text = malloc (40 + strlen (arg));
        if (pending_error_text)
1100        sprintf (pending_error_text, "E cannot close %s", arg);
        pending_error = errno;
        if (gzip_pid)
            waitpid (gzip_pid, (int *) 0, 0);
        return;
    }

    if (gzip_pid)
    {
        if (waitpid (gzip_pid, &gzip_status, 0) != gzip_pid)
1110        error (1, errno, "waiting for gunzip process %ld",
                   (long) gzip_pid);
        else if (gzip_status != 0)
            error (1, 0, "gunzip exited %d", gzip_status);
    }
}

/* Kopt for the next file sent in Modified or Is-modified.  */
static char *kopt;

1120 static void serve_modified PROTO ((char *));

static void
serve_modified (arg)
     char *arg;
{
    int size, status;
    char *size_text;
    char *mode_text;

1130    int gzipped = 0;

    /*
     * This used to return immediately if error_pending () was true.
     * However, that fails, because it causes each line of the file to
     * be echoed back to the client as an unrecognized command.  The
     * client isn't reading from the socket, so eventually both
     * processes block trying to write to the other.  Now, we try to
     * read the file if we can.
     */
1140
    status = buf_read_line (buf_from_net, &mode_text, (int *) NULL);
    if (status != 0)
    {
        if (status == -2)
            pending_error = ENOMEM;
        else
        {
            pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
1150            pending_error = ENOMEM;
            else
            {
                if (status == -1)
                    sprintf (pending_error_text,
                             "E end of file reading mode for %s", arg);
                else
                {
                    sprintf (pending_error_text,
                             "E error reading mode for %s", arg);
1160                pending_error = status;
                }
            }
        }
        return;
    }

    status = buf_read_line (buf_from_net, &size_text, (int *) NULL);
    if (status != 0)
```

```
    {
1170    if (status == -2)
            pending_error = ENOMEM;
        else
        {
            pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
                pending_error = ENOMEM;
            else
            {
                if (status == -1)
1180                sprintf (pending_error_text,
                             "E end of file reading size for %s", arg);
                else
                {
                    sprintf (pending_error_text,
                             "E error reading size for %s", arg);
                    pending_error = errno;
                }
            }
        }
1190    return;
    }
    if (size_text[0] == 'z')
    {
        gzipped = 1;
        size = atoi (size_text + 1);
    }
    else
        size = atoi (size_text);
    free (size_text);
1200

    if (error_pending ())
    {
        /* Now that we know the size, read and discard the file data.  */
        while (size > 0)
        {
            int status, nread;
            char *data;

            status = buf_read_data (buf_from_net, size, &data, &nread);
1210        if (status != 0)
                return;
            size -= nread;
        }
        return;
    }

    if (size >= 0)
    {
        receive_file (size, arg, gzipped);
1220    if (error_pending ()) return;
    }

    {
        int status = change_mode (arg, mode_text, 0);
        free (mode_text);
        if (status)
        {
            pending_error_text = malloc (40 + strlen (arg));
            if (pending_error_text)
1230            sprintf (pending_error_text,
                         "E cannot change mode for %s", arg);
            pending_error = status;
            return;
        }
    }

    /* Make sure that the Entries indicate the right kopt.  We probably
       could do this even in the non-kopt case and, I think, save a stat()
       call in time_stamp_server.  But for conservatism I'm leaving the
1240   non-kopt case alone.  */
    if (kopt != NULL)
        serve_is_modified (arg);
}

static void serve_remote_revision PROTO ((char *));

static void
serve_remote_revision (arg)
     char *arg;
1250 {
    int size, status;
    char *size_text;
    char *mode_text;
    char* rev_text;
    char* remote_rev_file;


    int gzipped = 0;
```

```
1260    printf ("M serve_remote_revision entered\n");

        /*
         * This used to return immediately if error_pending () was true.
         * However, that fails, because it causes each line of the file to
         * be echoed back to the client as an unrecognized command.  The
         * client isn't reading from the socket, so eventually both
         * processes block trying to write to the other.  Now, we try to
         * read the file if we can.
         */
1270
        status = buf_read_line (buf_from_net, &rev_text, (int *) NULL);
        if (status != 0)
        {
            if (status == -2)
                pending_error = ENOMEM;
            else
            {
                pending_error_text = malloc (80 + strlen (arg));
                if (pending_error_text == NULL)
1280                pending_error = ENOMEM;
                else
                {
                    if (status == -1)
                        sprintf (pending_error_text,
                                 "E end of file reading revision for %s", arg);
                    else
                    {
                        sprintf (pending_error_text,
                                 "E error reading revision for %s", arg);
1290                    pending_error = status;
                    }
                }
            }
            return;
        }

        printf ("M serve_remote_revision read rev (%s)\n", rev_text);

        status = buf_read_line (buf_from_net, &mode_text, (int *) NULL);
1300    if (status != 0)
        {
            if (status == -2)
                pending_error = ENOMEM;
            else
            {
                pending_error_text = malloc (80 + strlen (arg));
                if (pending_error_text == NULL)
                    pending_error = ENOMEM;
                else
1310            {
                    if (status == -1)
                        sprintf (pending_error_text,
                                 "E end of file reading mode for %s", arg);
                    else
                    {
                        sprintf (pending_error_text,
                                 "E error reading mode for %s", arg);
                        pending_error = status;
                    }
1320            }
            }
            return;
        }

        printf ("M serve_remote_revision read mode (%s)\n", mode_text);

        status = buf_read_line (buf_from_net, &size_text, (int *) NULL);
        if (status != 0)
        {
1330        if (status == -2)
                pending_error = ENOMEM;
            else
            {
                pending_error_text = malloc (80 + strlen (arg));
                if (pending_error_text == NULL)
                    pending_error = ENOMEM;
                else
                {
                    if (status == -1)
1340                    sprintf (pending_error_text,
                                 "E end of file reading size for %s", arg);
                    else
                    {
                        sprintf (pending_error_text,
                                 "E error reading size for %s", arg);
                        pending_error = errno;
                    }
                }
```

```
        }
        return;
    }
    if (size_text[0] == 'z')
    {
        gzipped = 1;
        size = atoi (size_text + 1);
    }
    else
        size = atoi (size_text);
    printf ("M serve_remote_revision read size (%s, %d)\n", size_text, size);

    free (size_text);

    if (error_pending ())
    {
        /* Now that we know the size, read and discard the file data.  */
        while (size > 0)
        {
            int status, nread;
            char *data;

            status = buf_read_data (buf_from_net, size, &data, &nread);
            if (status != 0)
                return;
            size -= nread;
        }
        return;
    }

    remote_rev_file = xmalloc (strlen (arg) + strlen (CVSADM_REMOTE_TMP) + strlen (rev_text) + 3);
    sprintf (remote_rev_file, "%s_%s_%s", CVSADM_REMOTE_TMP, rev_text, arg);

    if (size >= 0)
    {
        receive_file (size, remote_rev_file, gzipped);
        if (error_pending ()) return;
    }

    {
        int status = change_mode (remote_rev_file, mode_text, 0);
        free (mode_text);
        if (status)
        {
            pending_error_text = malloc (40 + strlen (remote_rev_file));
            if (pending_error_text)
                sprintf (pending_error_text,
                         "E cannot change mode for %s", remote_rev_file);
            pending_error = status;
            return;
        }
    }

    printf ("M serve_remote_revision returning\n");
}


static void
serve_enable_unchanged (arg)
     char *arg;
{
}

struct an_entry {
    struct an_entry *next;
    char *entry;
};

static struct an_entry *entries;

static void serve_unchanged PROTO ((char *));

static void
serve_unchanged (arg)
    char *arg;
{
    struct an_entry *p;
    char *name;
    char *cp;
    char *timefield;

    if (error_pending ())
        return;

    /* Rewrite entries file to have '=' in timestamp field.  */
    for (p = entries; p != NULL; p = p->next)
    {
        name = p->entry + 1;
        cp = strchr (name, '/');
        if (cp != NULL
```

```
            && strlen (arg) == cp − name
            && strncmp (arg, name, cp − name) == 0)
        {
            timefield = strchr (cp + 1, '/') + 1;
            if (*timefield != '=')
            {
                cp = timefield + strlen (timefield);
                cp[1] = '\0';
                while (cp > timefield)
                {
                    *cp = cp[−1];
                    −−cp;
                }
                *timefield = '=';
            }
            break;
        }
    }
}

static void
serve_is_modified (arg)
    char *arg;
{
    struct an_entry *p;
    char *name;
    char *cp;
    char *timefield;
    /* Have we found this file in "entries" yet.  */
    int found;

    if (error_pending ())
        return;

    /* Rewrite entries file to have 'M' in timestamp field.  */
    found = 0;
    for (p = entries; p != NULL; p = p->next)
    {
        name = p->entry + 1;
        cp = strchr (name, '/');
        if (cp != NULL
            && strlen (arg) == cp − name
            && strncmp (arg, name, cp − name) == 0)
        {
            timefield = strchr (cp + 1, '/') + 1;
            if (!(timefield[0] == 'M' && timefield[1] == '/'))
            {
                cp = timefield + strlen (timefield);
                cp[1] = '\0';
                while (cp > timefield)
                {
                    *cp = cp[−1];
                    −−cp;
                }
                *timefield = 'M';
            }
            if (kopt != NULL)
            {
                if (alloc_pending (strlen (name) + 80))
                    sprintf (pending_error_text,
                             "E protocol error: both Kopt and Entry for %s",
                             arg);
                free (kopt);
                kopt = NULL;
                return;
            }
            found = 1;
            break;
        }
    }
    if (!found)
    {
        /* We got Is-modified but no Entry.  Add a dummy entry.
           The "D" timestamp is what makes it a dummy.  */
        p = (struct an_entry *) malloc (sizeof (struct an_entry));
        if (p == NULL)
        {
            pending_error = ENOMEM;
            return;
        }
        p->entry = malloc (strlen (arg) + 80);
        if (p->entry == NULL)
        {
            pending_error = ENOMEM;
            free (p);
            return;
        }
        strcpy (p->entry, "/");
        strcat (p->entry, arg);
        strcat (p->entry, "//D/");
```

```
        if (kopt != NULL)
        {
            strcat (p->entry, kopt);
            free (kopt);
            kopt = NULL;
        }
        strcat (p->entry, "/");
        p->next = entries;
        entries = p;
    }
}

static void serve_entry PROTO ((char *));

static void
serve_entry (arg)
    char *arg;
{
    struct an_entry *p;
    char *cp;
    if (error_pending()) return;
    p = (struct an_entry *) malloc (sizeof (struct an_entry));
    if (p == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    /* Leave space for serve_unchanged to write '=' if it wants.  */
    cp = malloc (strlen (arg) + 2);
    if (cp == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (cp, arg);
    p->next = entries;
    p->entry = cp;
    entries = p;
}

static void serve_kopt PROTO ((char *));

static void
serve_kopt (arg)
    char *arg;
{
    if (error_pending ())
        return;

    if (kopt != NULL)
    {
        if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text,
                     "E protocol error: duplicate Kopt request: %s", arg);
        return;
    }

    /* Do some sanity checks.  In particular, that it is not too long.
       This lets the rest of the code not worry so much about buffer
       overrun attacks.  Probably should call RCS_check_kflag here,
       but that would mean changing RCS_check_kflag to handle errors
       other than via exit(), fprintf(), and such.  */
    if (strlen (arg) > 10)
    {
        if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text,
                     "E protocol error: invalid Kopt request: %s", arg);
        return;
    }

    kopt = malloc (strlen (arg) + 1);
    if (kopt == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (kopt, arg);
}

static void
server_write_entries ()
{
    FILE *f;
    struct an_entry *p;
    struct an_entry *q;

    if (entries == NULL)
        return;

    f = NULL;
```

```
    /* Note that we free all the entries regardless of errors.  */
    if (!error_pending ())
    {
        /* We open in append mode because we don't want to clobber an
           existing Entries file.  If we are checking out a module
           which explicitly lists more than one file in a particular
           directory, then we will wind up calling
           server_write_entries for each such file.  */
        f = CVS_FOPEN (CVSADM_ENT, "a");
        if (f == NULL)
        {
            pending_error = errno;
            if (alloc_pending (80 + strlen (CVSADM_ENT)))
                sprintf (pending_error_text, "E cannot open %s", CVSADM_ENT);
        }
    }
    for (p = entries; p != NULL;)
    {
        if (!error_pending ())
        {
            if (fprintf (f, "%s\n", p->entry) < 0)
            {
                pending_error = errno;
                if (alloc_pending (80 + strlen(CVSADM_ENT)))
                    sprintf (pending_error_text,
                             "E cannot write to %s", CVSADM_ENT);
            }
        }
        free (p->entry);
        q = p->next;
        free (p);
        p = q;
    }
    entries = NULL;
    if (f != NULL && fclose (f) == EOF && !error_pending ())
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_ENT)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_ENT);
    }
}

struct notify_note {
    /* Directory in which this notification happens.  malloc'd*/
    char *dir;

    /* malloc'd.  */
    char *filename;

    /* The following three all in one malloc'd block, pointed to by TYPE.
       Each '\0' terminated.  */
    /* "E" or "U".  */
    char *type;
    /* time+host+dir */
    char *val;
    char *watches;

    struct notify_note *next;
};

static struct notify_note *notify_list;
/* Used while building list, to point to the last node that already exists.  */
static struct notify_note *last_node;

static void serve_notify PROTO ((char *));

static void
serve_notify (arg)
    char *arg;
{
    struct notify_note *new;
    char *data;
    int status;

    if (error_pending ()) return;

    new = (struct notify_note *) malloc (sizeof (struct notify_note));
    if (new == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    if (dir_name == NULL)
        goto error;
    new->dir = malloc (strlen (dir_name) + 1);
    if (new->dir == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
```

```
    strcpy (new->dir, dir_name);
    new->filename = malloc (strlen (arg) + 1);
    if (new->filename == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (new->filename, arg);

    status = buf_read_line (buf_from_net, &data, (int *) NULL);
    if (status != 0)
    {
        if (status == -2)
            pending_error = ENOMEM;
        else
        {
            pending_error_text = malloc (80 + strlen (arg));
            if (pending_error_text == NULL)
                pending_error = ENOMEM;
            else
            {
                if (status == -1)
                    sprintf (pending_error_text,
                             "E end of file reading notification for %s", arg);
                else
                {
                    sprintf (pending_error_text,
                             "E error reading notification for %s", arg);
                    pending_error = status;
                }
            }
        }
    }
    else
    {
        char *cp;

        new->type = data;
        if (data[1] != '\t')
            goto error;
        data[1] = '\0';
        cp = data + 2;
        new->val = cp;
        cp = strchr (cp, '\t');
        if (cp == NULL)
            goto error;
        *cp++ = '+';
        cp = strchr (cp, '\t');
        if (cp == NULL)
            goto error;
        *cp++ = '+';
        cp = strchr (cp, '\t');
        if (cp == NULL)
            goto error;
        *cp++ = '\0';
        new->watches = cp;
        /* If there is another tab, ignore everything after it,
           for future expansion.  */
        cp = strchr (cp, '\t');
        if (cp != NULL)
        {
            *cp = '\0';
        }

        new->next = NULL;

        if (last_node == NULL)
        {
            notify_list = new;
        }
        else
            last_node->next = new;
        last_node = new;
    }
    return;
  error:
    pending_error_text = malloc (40);
    if (pending_error_text)
        strcpy (pending_error_text,
                "E Protocol error; misformed Notify request");
    pending_error = 0;
    return;
}

/* Process all the Notify requests that we have stored up.  Returns 0
   if successful, if not prints error message (via error()) and
   returns negative value.  */
static int
server_notify ()
{
```

```
    struct notify_note *p;
1800    char *repos;

    while (notify_list != NULL)
    {
        if ( CVS_CHDIR (notify_list->dir) < 0)
        {
            error (0, errno, "cannot change to %s", notify_list->dir);
            return -1;
        }
        repos = Name_Repository (NULL, NULL);
1810
        lock_dir_for_write (repos);

        fileattr_startdir (repos);

        notify_do (*notify_list->type, notify_list->filename, getcaller(),
                   notify_list->val, notify_list->watches, repos);

        buf_output0 (buf_to_net, "Notified ");
        {
1820        char *dir = notify_list->dir + strlen (server_temp_dir) + 1;
            if (dir[0] == '\0')
                buf_append_char (buf_to_net, '.');
            else
                buf_output0 (buf_to_net, dir);
            buf_append_char (buf_to_net, '/');
            buf_append_char (buf_to_net, '\n');
        }
        buf_output0 (buf_to_net, repos);
        buf_append_char (buf_to_net, '/');
1830    buf_output0 (buf_to_net, notify_list->filename);
        buf_append_char (buf_to_net, '\n');

        p = notify_list->next;
        free (notify_list->filename);
        free (notify_list->dir);
        free (notify_list->type);
        free (notify_list);
        notify_list = p;

1840    fileattr_write ();
        fileattr_free ();

        Lock_Cleanup ();
    }

    /* The code used to call fflush (stdout) here, but that is no
       longer necessary.  The data is now buffered in buf_to_net,
       which will be flushed by the caller, do_cvs_command.  */

1850    return 0;
}

static int argument_count;
static char **argument_vector;
static int argument_vector_size;

static void
serve_argument (arg)
     char *arg;
1860 {
    char *p;

    if (error_pending()) return;

    if (argument_vector_size <= argument_count)
    {
        argument_vector_size *= 2;
        argument_vector =
            (char **) realloc ((char *)argument_vector,
1870                           argument_vector_size * sizeof (char *));
        if (argument_vector == NULL)
        {
            pending_error = ENOMEM;
            return;
        }
    }
    p = malloc (strlen (arg) + 1);
    if (p == NULL)
    {
1880    pending_error = ENOMEM;
        return;
    }
    strcpy (p, arg);
    argument_vector[argument_count++] = p;
}

static void
serve_argumentx (arg)
```

```
    char *arg;
{
    char *p;

    if (error_pending()) return;

    p = argument_vector[argument_count - 1];
    p = realloc (p, strlen (p) + 1 + strlen (arg) + 1);
    if (p == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcat (p, "\n");
    strcat (p, arg);
    argument_vector[argument_count - 1] = p;
}

static void
serve_global_option (arg)
    char *arg;
{
    if (arg[0] != '-' || arg[1] == '\0' || arg[2] != '\0')
    {
    error_return:
        if (alloc_pending (strlen (arg) + 80))
            sprintf (pending_error_text,
                     "E Protocol error: bad global option %s",
                     arg);
        return;
    }
    switch (arg[1])
    {
        case 'n':
            noexec = 1;
            break;
        case 'q':
            quiet = 1;
            break;
        case 'r':
            cvswrite = 0;
            break;
        case 'Q':
            really_quiet = 1;
            break;
        case 'l':
            logoff = 1;
            break;
        case 't':
            trace = 1;
            break;
        default:
            goto error_return;
    }
}

static void
serve_set (arg)
    char *arg;
{
    /* FIXME: This sends errors immediately (I think); they should be
       put into pending_error.  */
    variable_set (arg);
}

#ifdef ENCRYPTION

#ifdef HAVE_KERBEROS

static void
serve_kerberos_encrypt (arg)
    char *arg;
{
    /* All future communication with the client will be encrypted.  */

    buf_to_net = krb_encrypt_buffer_initialize (buf_to_net, 0, sched,
                                                kblock,
                                                buf_to_net->memory_error);
    buf_from_net = krb_encrypt_buffer_initialize (buf_from_net, 1, sched,
                                                  kblock,
                                                  buf_from_net->memory_error);
}

#endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI

static void
serve_gssapi_encrypt (arg)
    char *arg;
```

```
{
    if (cvs_gssapi_wrapping)
    {
        /* We're already using a gssapi_wrap buffer for stream
           authentication.  Flush everything we've output so far, and
           turn on encryption for future data.  On the input side, we
           should only have unwrapped as far as the Gssapi-encrypt
           command, so future unwrapping will become encrypted.  */
        buf_flush (buf_to_net, 1);
        cvs_gssapi_encrypt = 1;
        return;
    }

    /* All future communication with the client will be encrypted.  */

    cvs_gssapi_encrypt = 1;

    buf_to_net = cvs_gssapi_wrap_buffer_initialize (buf_to_net, 0,
                                                    gcontext,
                                                    buf_to_net->memory_error);
    buf_from_net = cvs_gssapi_wrap_buffer_initialize (buf_from_net, 1,
                                                      gcontext,
                                                      buf_from_net->memory_error);

    cvs_gssapi_wrapping = 1;
}

#endif /* HAVE_GSSAPI */

#endif /* ENCRYPTION */

#ifdef HAVE_GSSAPI

static void
serve_gssapi_authenticate (arg)
    char *arg;
{
    if (cvs_gssapi_wrapping)
    {
        /* We're already using a gssapi_wrap buffer for encryption.
           That includes authentication, so we don't have to do
           anything further.  */
        return;
    }

    buf_to_net = cvs_gssapi_wrap_buffer_initialize (buf_to_net, 0,
                                                    gcontext,
                                                    buf_to_net->memory_error);
    buf_from_net = cvs_gssapi_wrap_buffer_initialize (buf_from_net, 1,
                                                      gcontext,
                                                      buf_from_net->memory_error);

    cvs_gssapi_wrapping = 1;
}

#endif /* HAVE_GSSAPI */

#ifdef SERVER_FLOWCONTROL
/* The maximum we'll queue to the remote client before blocking.  */
# ifndef SERVER_HI_WATER
#  define SERVER_HI_WATER (2 * 1024 * 1024)
# endif /* SERVER_HI_WATER */
/* When the buffer drops to this, we restart the child */
# ifndef SERVER_LO_WATER
#  define SERVER_LO_WATER (1 * 1024 * 1024)
# endif /* SERVER_LO_WATER */

static int set_nonblock_fd PROTO((int));

/*
 * Set buffer BUF to non-blocking I/O.  Returns 0 for success or errno
 * code.
 */

static int
set_nonblock_fd (fd)
    int fd;
{
    int flags;

    flags = fcntl (fd, F_GETFL, 0);
    if (flags < 0)
        return errno;
    if (fcntl (fd, F_SETFL, flags | O_NONBLOCK) < 0)
        return errno;
    return 0;
}

#endif /* SERVER_FLOWCONTROL */
```

```
static void serve_questionable PROTO((char *));

static void
serve_questionable (arg)
    char *arg;
{
    static int initted;

    if (!initted)
    {
        /* Pick up ignores from CVSROOTADM_IGNORE, $HOME/.cvsignore on server,
           and CVSIGNORE on server.  */
        ign_setup ();
        initted = 1;
    }

    if (dir_name == NULL)
    {
        buf_output0 (buf_to_net, "E Protocol error: 'Directory' missing");
        return;
    }

    if (!ign_name (arg))
    {
        char *update_dir;

        buf_output (buf_to_net, "M ? ", 4);
        update_dir = dir_name + strlen (server_temp_dir) + 1;
        if (!(update_dir[0] == '.' && update_dir[1] == '\0'))
        {
            buf_output0 (buf_to_net, update_dir);
            buf_output (buf_to_net, "/", 1);
        }
        buf_output0 (buf_to_net, arg);
        buf_output (buf_to_net, "\n", 1);
    }
}

static void serve_case PROTO ((char *));

static void
serve_case (arg)
    char *arg;
{
    ign_case = 1;
}

static struct buffer *protocol;

/* This is the output which we are saving up to send to the server, in the
   child process.  We will push it through, via the 'protocol' buffer, when
   we have a complete line.  */
static struct buffer *saved_output;
/* Likewise, but stuff which will go to stderr.  */
static struct buffer *saved_outerr;

static void
protocol_memory_error (buf)
    struct buffer *buf;
{
    error (1, ENOMEM, "Virtual memory exhausted");
}

/*
 * Process IDs of the subprocess, or negative if that subprocess
 * does not exist.
 */
static pid_t command_pid;

static void
outbuf_memory_error (buf)
    struct buffer *buf;
{
    static const char msg[] = "E Fatal server error\n\
error ENOMEM Virtual memory exhausted.\n";
    if (command_pid > 0)
        kill (command_pid, SIGTERM);

    /*
     * We have arranged things so that printing this now either will
     * be legal, or the "E fatal error" line will get glommed onto the
     * end of an existing "E" or "M" response.
     */

    /* If this gives an error, not much we could do.  syslog() it?  */
    write (STDOUT_FILENO, msg, sizeof (msg) - 1);
    error_exit ();
}

static void
```

```
input_memory_error (buf)
     struct buffer *buf;
{
    outbuf_memory_error (buf);
}



/* If command is legal, return 1.
 * Else if command is illegal and croak_on_illegal is set, then die.
 * Else just return 0 to indicate that command is illegal.
 */
static int
check_command_legal_p (cmd_name)
    char *cmd_name;
{
    /* Right now, only pserver notices illegal commands – namely,
     * write attempts by a read-only user.  Therefore, if CVS_Username
     * is not set, this just returns 1, because CVS_Username unset
     * means pserver is not active.
     */
#ifdef AUTH_SERVER_SUPPORT
    if (CVS_Username == NULL)
        return 1;

    if (lookup_command_attribute (cmd_name) & CVS_CMD_MODIFIES_REPOSITORY)
    {
        /* This command has the potential to modify the repository, so
         * we check if the user have permission to do that.
         *
         * (Only relevant for remote users – local users can do
         * whatever normal Unix file permissions allow them to do.)
         *
         * The decision method:
         *
         *    If $CVSROOT/CVSADMROOT_READERS exists and user is listed
         *    in it, then read-only access for user.
         *
         *    Or if $CVSROOT/CVSADMROOT_WRITERS exists and user NOT
         *    listed in it, then also read-only access for user.
         *
         *    Else read-write access for user.
         */

        char *linebuf = NULL;
        int num_red = 0;
        size_t linebuf_len = 0;
        char *fname;
        size_t flen;
        FILE *fp;
        int found_it = 0;

        /* else */
        flen = strlen (CVSroot_directory)
               + strlen (CVSROOTADM)
               + strlen (CVSROOTADM_READERS)
               + 3;

        fname = xmalloc (flen);
        (void) sprintf (fname, "%s/%s/%s", CVSroot_directory,
                        CVSROOTADM, CVSROOTADM_READERS);

        fp = fopen (fname, "r");

        if (fp == NULL)
        {
            if (!existence_error (errno))
            {
                /* Need to deny access, so that attackers can't fool
                   us with some sort of denial of service attack.  */
                error (0, errno, "cannot open %s", fname);
                free (fname);
                return 0;
            }
        }
        else  /* successfully opened readers file */
        {
            while ((num_red = getline (&linebuf, &linebuf_len, fp)) >= 0)
            {
                /* Hmmm, is it worth importing my own readline
                   library into CVS?  It takes care of chopping
                   leading and trailing whitespace, "#" comments, and
                   newlines automatically when so requested.  Would
                   save some code here...  -kff */

                /* Chop newline by hand, for strcmp()'s sake. */
                if (linebuf[num_red - 1] == '\n')
                    linebuf[num_red - 1] = '\0';

                if (strcmp (linebuf, CVS_Username) == 0)
```

```
                goto handle_illegal;
        }
        if (num_red < 0 && !feof (fp))
            error (0, errno, "cannot read %s", fname);

        /* If not listed specifically as a reader, then this user
           has write access by default unless writers are also
           specified in a file . */
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", fname);
    }
    free (fname);

    /* Now check the writers file.  */

    flen = strlen (CVSroot_directory)
           + strlen (CVSROOTADM)
           + strlen (CVSROOTADM_WRITERS)
           + 3;

    fname = xmalloc (flen);
    (void) sprintf (fname, "%s/%s/%s", CVSroot_directory,
                    CVSROOTADM, CVSROOTADM_WRITERS);

    fp = fopen (fname, "r");

    if (fp == NULL)
    {
        if (linebuf)
            free (linebuf);
        if (existence_error (errno))
        {
            /* Writers file does not exist, so everyone is a writer,
               by default.  */
            free (fname);
            return 1;
        }
        else
        {
            /* Need to deny access, so that attackers can't fool
               us with some sort of denial of service attack.  */
            error (0, errno, "cannot read %s", fname);
            free (fname);
            return 0;
        }
    }

    found_it = 0;
    while ((num_red = getline (&linebuf, &linebuf_len, fp)) >= 0)
    {
        /* Chop newline by hand, for strcmp()'s sake. */
        if (linebuf[num_red - 1] == '\n')
            linebuf[num_red - 1] = '\0';

        if (strcmp (linebuf, CVS_Username) == 0)
        {
            found_it = 1;
            break;
        }
    }
    if (num_red < 0 && !feof (fp))
        error (0, errno, "cannot read %s", fname);

    if (found_it)
    {
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", fname);
        if (linebuf)
            free (linebuf);
        free (fname);
        return 1;
    }
    else   /* writers file exists, but this user not listed in it */
    {
    handle_illegal:
        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", fname);
        if (linebuf)
            free (linebuf);
        free (fname);
        return 0;
    }
}
#endif /* AUTH_SERVER_SUPPORT */

    /* If ever reach end of this function, command must be legal. */
    return 1;
}
```

```
2340  /* Execute COMMAND in a subprocess with the approriate funky things done.  */

      static struct fd_set_wrapper { fd_set fds; } command_fds_to_drain;
      static int max_command_fd;

      #ifdef SERVER_FLOWCONTROL
      static int flowcontrol_pipe[2];
      #endif /* SERVER_FLOWCONTROL */

      static void
2350  do_cvs_command (cmd_name, command)
          char *cmd_name;
          int (*command) PROTO((int argc, char **argv));
      {
          /*
           * The following file descriptors are set to -1 if that file is not
           * currently open.
           */

          /* Data on these pipes is a series of '\n'-terminated lines.  */
2360      int stdout_pipe[2];
          int stderr_pipe[2];

          /*
           * Data on this pipe is a series of counted (see buf_send_counted)
           * packets.  Each packet must be processed atomically (i.e. not
           * interleaved with data from stdout_pipe or stderr_pipe).
           */
          int protocol_pipe[2];

2370      int dev_null_fd = -1;

          int errs;

          command_pid = -1;
          stdout_pipe[0] = -1;
          stdout_pipe[1] = -1;
          stderr_pipe[0] = -1;
          stderr_pipe[1] = -1;
          protocol_pipe[0] = -1;
2380      protocol_pipe[1] = -1;

          server_write_entries ();

          if (print_pending_error ())
              goto free_args_and_return;

          /* Global 'command_name' is probably "server" right now - only
             serve_export() sets it to anything else.  So we will use local
             parameter 'cmd_name' to determine if this command is legal for
2390         this user.  */
          if (!check_command_legal_p (cmd_name))
          {
              buf_output0 (buf_to_net, "E ");
              buf_output0 (buf_to_net, program_name);
              buf_output0 (buf_to_net, " [server aborted]: \"");
              buf_output0 (buf_to_net, cmd_name);
              buf_output0 (buf_to_net, "\" requires write access to the repository\n\
      error \n");
              goto free_args_and_return;
2400      }

          (void) server_notify ();

          /*
           * We use a child process which actually does the operation.  This
           * is so we can intercept its standard output.  Even if all of CVS
           * were written to go to some special routine instead of writing
           * to stdout or stderr, we would still need to do the same thing
           * for the RCS commands.
2410       */

          if (pipe (stdout_pipe) < 0)
          {
              print_error (errno);
              goto error_exit;
          }
          if (pipe (stderr_pipe) < 0)
          {
              print_error (errno);
2420          goto error_exit;
          }
          if (pipe (protocol_pipe) < 0)
          {
              print_error (errno);
              goto error_exit;
          }
      #ifdef SERVER_FLOWCONTROL
          if (pipe (flowcontrol_pipe) < 0)
```

```
    {
2430        print_error (errno);
        goto error_exit;
    }
    set_nonblock_fd (flowcontrol_pipe[0]);
    set_nonblock_fd (flowcontrol_pipe[1]);
#endif /* SERVER_FLOWCONTROL */

    dev_null_fd = CVS_OPEN (DEVNULL, O_RDONLY);
    if (dev_null_fd < 0)
    {
2440        print_error (errno);
        goto error_exit;
    }

    /* We shouldn't have any partial lines from cvs_output and
       cvs_outerr, but we handle them here in case there is a bug.  */
    /* FIXME: appending a newline, rather than using "MT" as we
       do in the child process, is probably not really a very good
       way to "handle" them.  */
    if (! buf_empty_p (saved_output))
2450    {
        buf_append_char (saved_output, '\n');
        buf_copy_lines (buf_to_net, saved_output, 'M');
    }
    if (! buf_empty_p (saved_outerr))
    {
        buf_append_char (saved_outerr, '\n');
        buf_copy_lines (buf_to_net, saved_outerr, 'E');
    }

2460    /* Flush out any pending data.  */
    buf_flush (buf_to_net, 1);

    /* Don't use vfork; we're not going to exec().  */
    command_pid = fork ();
    if (command_pid < 0)
    {
        print_error (errno);
        goto error_exit;
    }
2470    if (command_pid == 0)
    {
        int exitstatus;

        /* Since we're in the child, and the parent is going to take
           care of packaging up our error messages, we can clear this
           flag.  */
        error_use_protocol = 0;

        protocol = fd_buffer_initialize (protocol_pipe[1], 0,
2480                                         protocol_memory_error);

        /* At this point we should no longer be using buf_to_net and
           buf_from_net.  Instead, everything should go through
           protocol.  */
        buf_to_net = NULL;
        buf_from_net = NULL;

        /* These were originally set up to use outbuf_memory_error.
           Since we're now in the child, we should use the simpler
2490       protocol_memory_error function.  */
        saved_output->memory_error = protocol_memory_error;
        saved_outerr->memory_error = protocol_memory_error;

        if (dup2 (dev_null_fd, STDIN_FILENO) < 0)
            error (1, errno, "can't set up pipes");
        if (dup2 (stdout_pipe[1], STDOUT_FILENO) < 0)
            error (1, errno, "can't set up pipes");
        if (dup2 (stderr_pipe[1], STDERR_FILENO) < 0)
            error (1, errno, "can't set up pipes");
2500    close (stdout_pipe[0]);
        close (stderr_pipe[0]);
        close (protocol_pipe[0]);
#ifdef SERVER_FLOWCONTROL
        close (flowcontrol_pipe[1]);
#endif /* SERVER_FLOWCONTROL */

        /*
         * Set this in .bashrc if you want to give yourself time to attach
         * to the subprocess with a debugger.
2510     */
        if (getenv ("CVS_SERVER_SLEEP"))
        {
            int secs = atoi (getenv ("CVS_SERVER_SLEEP"));
            sleep (secs);
        }

        exitstatus = (*command) (argument_count, argument_vector);
```

```
        /* Output any partial lines.  If the client doesn't support
           "MT", we just throw out the partial line, like old versions
           of CVS did, since the protocol can't support this.  */
        if (supported_response ("MT") && ! buf_empty_p (saved_output))
        {
            buf_output0 (protocol, "MT text ");
            buf_append_buffer (protocol, saved_output);
            buf_output (protocol, "\n", 1);
            buf_send_counted (protocol);
        }
        /* For now we just discard partial lines on stderr.  I suspect
           that CVS can't write such lines unless there is a bug.  */

        /*
         * When we exit, that will close the pipes, giving an EOF to
         * the parent.
         */
        exit (exitstatus);
    }

    /* OK, sit around getting all the input from the child.  */
    {
        struct buffer *stdoutbuf;
        struct buffer *stderrbuf;
        struct buffer *protocol_inbuf;
        /* Number of file descriptors to check in select ().  */
        int num_to_check;
        int count_needed = 0;
#ifdef SERVER_FLOWCONTROL
        int have_flowcontrolled = 0;
#endif /* SERVER_FLOWCONTROL */

        FD_ZERO (&command_fds_to_drain.fds);
        num_to_check = stdout_pipe[0];
        FD_SET (stdout_pipe[0], &command_fds_to_drain.fds);
        if (stderr_pipe[0] > num_to_check)
          num_to_check = stderr_pipe[0];
        FD_SET (stderr_pipe[0], &command_fds_to_drain.fds);
        if (protocol_pipe[0] > num_to_check)
          num_to_check = protocol_pipe[0];
        FD_SET (protocol_pipe[0], &command_fds_to_drain.fds);
        if (STDOUT_FILENO > num_to_check)
          num_to_check = STDOUT_FILENO;
        max_command_fd = num_to_check;
        /*
         * File descriptors are numbered from 0, so num_to_check needs to
         * be one larger than the largest descriptor.
         */
        ++num_to_check;
        if (num_to_check > FD_SETSIZE)
        {
            buf_output0 (buf_to_net,
                         "E internal error: FD_SETSIZE not big enough.\n\
error \n");
            goto error_exit;
        }

        stdoutbuf = fd_buffer_initialize (stdout_pipe[0], 1,
                                          input_memory_error);

        stderrbuf = fd_buffer_initialize (stderr_pipe[0], 1,
                                          input_memory_error);

        protocol_inbuf = fd_buffer_initialize (protocol_pipe[0], 1,
                                               input_memory_error);

        set_nonblock (buf_to_net);
        set_nonblock (stdoutbuf);
        set_nonblock (stderrbuf);
        set_nonblock (protocol_inbuf);

        if (close (stdout_pipe[1]) < 0)
        {
            print_error (errno);
            goto error_exit;
        }
        stdout_pipe[1] = -1;

        if (close (stderr_pipe[1]) < 0)
        {
            print_error (errno);
            goto error_exit;
        }
        stderr_pipe[1] = -1;

        if (close (protocol_pipe[1]) < 0)
        {
            print_error (errno);
            goto error_exit;
        }
```

```
        protocol_pipe[1] = -1;

#ifdef SERVER_FLOWCONTROL
        if (close (flowcontrol_pipe[0]) < 0)
        {
            print_error (errno);
            goto error_exit;
        }
        flowcontrol_pipe[0] = -1;
#endif /* SERVER_FLOWCONTROL */

        if (close (dev_null_fd) < 0)
        {
            print_error (errno);
            goto error_exit;
        }
        dev_null_fd = -1;

        while (stdout_pipe[0] >= 0
               || stderr_pipe[0] >= 0
               || protocol_pipe[0] >= 0)
        {
            fd_set readfds;
            fd_set writefds;
            int numfds;
#ifdef SERVER_FLOWCONTROL
            int bufmemsize;

            /*
             * See if we are swamping the remote client and filling our VM.
             * Tell child to hold off if we do.
             */
            bufmemsize = buf_count_mem (buf_to_net);
            if (!have_flowcontrolled && (bufmemsize > SERVER_HI_WATER))
            {
                if (write(flowcontrol_pipe[1], "S", 1) == 1)
                    have_flowcontrolled = 1;
            }
            else if (have_flowcontrolled && (bufmemsize < SERVER_LO_WATER))
            {
                if (write(flowcontrol_pipe[1], "G", 1) == 1)
                    have_flowcontrolled = 0;
            }
#endif /* SERVER_FLOWCONTROL */

            FD_ZERO (&readfds);
            FD_ZERO (&writefds);
            if (! buf_empty_p (buf_to_net))
                FD_SET (STDOUT_FILENO, &writefds);

            if (stdout_pipe[0] >= 0)
            {
                FD_SET (stdout_pipe[0], &readfds);
            }
            if (stderr_pipe[0] >= 0)
            {
                FD_SET (stderr_pipe[0], &readfds);
            }
            if (protocol_pipe[0] >= 0)
            {
                FD_SET (protocol_pipe[0], &readfds);
            }

            /* This process of selecting on the three pipes means that
               we might not get output in the same order in which it
               was written, thus producing the well-known
               "out-of-order" bug.  If the child process uses
               cvs_output and cvs_outerr, it will send everything on
               the protocol_pipe and avoid this problem, so the
               solution is to use cvs_output and cvs_outerr in the
               child process.  */
            do {
                /* This used to select on exceptions too, but as far
                   as I know there was never any reason to do that and
                   SCO doesn't let you select on exceptions on pipes.  */
                numfds = select (num_to_check, &readfds, &writefds,
                                 (fd_set *)0, (struct timeval *)NULL);
                if (numfds < 0
                    && errno != EINTR)
                {
                    print_error (errno);
                    goto error_exit;
                }
            } while (numfds < 0);

            if (FD_ISSET (STDOUT_FILENO, &writefds))
            {
                /* What should we do with errors?  syslog() them?  */
                buf_send_output (buf_to_net);
            }
```

```
2700        if (stdout_pipe[0] >= 0
                && (FD_ISSET (stdout_pipe[0], &readfds)))
            {
                int status;

                status = buf_input_data (stdoutbuf, (int *) NULL);

                buf_copy_lines (buf_to_net, stdoutbuf, 'M');

                if (status == -1)
2710                stdout_pipe[0] = -1;
                else if (status > 0)
                {
                    print_error (status);
                    goto error_exit;
                }

                /* What should we do with errors?  syslog() them?  */
                buf_send_output (buf_to_net);
            }
2720
            if (stderr_pipe[0] >= 0
                && (FD_ISSET (stderr_pipe[0], &readfds)))
            {
                int status;

                status = buf_input_data (stderrbuf, (int *) NULL);

                buf_copy_lines (buf_to_net, stderrbuf, 'E');

2730            if (status == -1)
                    stderr_pipe[0] = -1;
                else if (status > 0)
                {
                    print_error (status);
                    goto error_exit;
                }

                /* What should we do with errors?  syslog() them?  */
                buf_send_output (buf_to_net);
2740        }

            if (protocol_pipe[0] >= 0
                && (FD_ISSET (protocol_pipe[0], &readfds)))
            {
                int status;
                int count_read;
                int special;

                status = buf_input_data (protocol_inbuf, &count_read);
2750
                if (status == -1)
                    protocol_pipe[0] = -1;
                else if (status > 0)
                {
                    print_error (status);
                    goto error_exit;
                }

                /*
2760             * We only call buf_copy_counted if we have read
                 * enough bytes to make it worthwhile.  This saves us
                 * from continually recounting the amount of data we
                 * have.
                 */
                count_needed -= count_read;
                while (count_needed <= 0)
                {
                    count_needed = buf_copy_counted (buf_to_net,
                                                     protocol_inbuf,
2770                                                 &special);

                    /* What should we do with errors?  syslog() them?  */
                    buf_send_output (buf_to_net);

                    /* If SPECIAL got set to -1, it means that the child
                       wants us to flush the pipe.  We don't want to block
                       on the network, but we flush what we can.  If the
                       client supports the 'F' command, we send it.  */
                    if (special == -1)
2780                {
                        if (supported_response ("F"))
                        {
                            buf_append_char (buf_to_net, 'F');
                            buf_append_char (buf_to_net, '\n');
                        }

                        cvs_flusherr ();
                    }
```

```
                }
2790        }
        }

        /*
         * OK, we've gotten EOF on all the pipes.  If there is
         * anything left on stdoutbuf or stderrbuf (this could only
         * happen if there was no trailing newline), send it over.
         */
        if (! buf_empty_p (stdoutbuf))
        {
2800        buf_append_char (stdoutbuf, '\n');
            buf_copy_lines (buf_to_net, stdoutbuf, 'M');
        }
        if (! buf_empty_p (stderrbuf))
        {
            buf_append_char (stderrbuf, '\n');
            buf_copy_lines (buf_to_net, stderrbuf, 'E');
        }
        if (! buf_empty_p (protocol_inbuf))
            buf_output0 (buf_to_net,
2810                     "E Protocol error: uncounted data discarded\n");

        errs = 0;

        while (command_pid > 0)
        {
            int status;
            pid_t waited_pid;
            waited_pid = waitpid (command_pid, &status, 0);
            if (waited_pid < 0)
2820        {
                /*
                 * Intentionally ignoring EINTR.  Other errors
                 * "can't happen".
                 */
                continue;
            }

            if (WIFEXITED (status))
                errs += WEXITSTATUS (status);
2830        else
            {
                int sig = WTERMSIG (status);
                char buf[50];
                /*
                 * This is really evil, because signals might be numbered
                 * differently on the two systems.  We should be using
                 * signal names (either of the "Terminated" or the "SIGTERM"
                 * variety).  But cvs doesn't currently use libiberty...we
                 * could roll our own....  FIXME.
2840             */
                buf_output0 (buf_to_net, "E Terminated with fatal signal ");
                sprintf (buf, "%d\n", sig);
                buf_output0 (buf_to_net, buf);

                /* Test for a core dump.  Is this portable?  */
                if (status & 0x80)
                {
                    buf_output0 (buf_to_net, "E Core dumped; preserving ");
                    buf_output0 (buf_to_net, orig_server_temp_dir);
2850                buf_output0 (buf_to_net, " on server.\n\
E CVS locks may need cleaning up.\n");
                    dont_delete_temp = 1;
                }
                ++errs;
            }
            if (waited_pid == command_pid)
                command_pid = -1;
        }

2860    /*
         * OK, we've waited for the child.  By now all CVS locks are free
         * and it's OK to block on the network.
         */
        set_block (buf_to_net);
        buf_flush (buf_to_net, 1);
    }

    if (errs)
        /* We will have printed an error message already.  */
2870    buf_output0 (buf_to_net, "error  \n");
    else
        buf_output0 (buf_to_net, "ok\n");
    goto free_args_and_return;

 error_exit:
    if (command_pid > 0)
        kill (command_pid, SIGTERM);
```

```
    while (command_pid > 0)
2880    {
            pid_t waited_pid;
            waited_pid = waitpid (command_pid, (int *) 0, 0);
            if (waited_pid < 0 && errno == EINTR)
                continue;
            if (waited_pid == command_pid)
                command_pid = -1;
        }

        close (dev_null_fd);
2890    close (protocol_pipe[0]);
        close (protocol_pipe[1]);
        close (stderr_pipe[0]);
        close (stderr_pipe[1]);
        close (stdout_pipe[0]);
        close (stdout_pipe[1]);

     free_args_and_return:
        /* Now free the arguments.  */
        {
2900        /* argument_vector[0] is a dummy argument, we don't mess with it.  */
            char **cp;
            for (cp = argument_vector + 1;
                 cp < argument_vector + argument_count;
                 ++cp)
                free (*cp);

            argument_count = 1;
        }

2910    /* Flush out any data not yet sent.  */
        set_block (buf_to_net);
        buf_flush (buf_to_net, 1);

        return;
    }

    #ifdef SERVER_FLOWCONTROL
    /*
     * Called by the child at convenient points in the server's execution for
2920 * the server child to block.. ie: when it has no locks active.
     */
    void
    server_pause_check()
    {
        int paused = 0;
        char buf[1];

        while (read (flowcontrol_pipe[0], buf, 1) == 1)
        {
2930        if (*buf == 'S')    /* Stop */
                paused = 1;
            else if (*buf == 'G') /* Go */
                paused = 0;
            else
                return;         /* ??? */
        }
        while (paused) {
            int numfds, numtocheck;
            fd_set fds;
2940
            FD_ZERO (&fds);
            FD_SET (flowcontrol_pipe[0], &fds);
            numtocheck = flowcontrol_pipe[0] + 1;

            do {
                numfds = select (numtocheck, &fds, (fd_set *)0,
                                 (fd_set *)0, (struct timeval *)NULL);
                if (numfds < 0
                    && errno != EINTR)
2950            {
                    print_error (errno);
                    return;
                }
            } while (numfds < 0);

            if (FD_ISSET (flowcontrol_pipe[0], &fds))
            {
                int got;

2960            while ((got = read (flowcontrol_pipe[0], buf, 1)) == 1)
                {
                    if (*buf == 'S')    /* Stop */
                        paused = 1;
                    else if (*buf == 'G') /* Go */
                        paused = 0;
                    else
                        return;         /* ??? */
                }
```

```
2970            /* This assumes that we are using BSD or POSIX nonblocking
                   I/O.  System V nonblocking I/O returns zero if there is
                   nothing to read.  */
                if (got == 0)
                    error (1, 0, "flow control EOF");
                if (got < 0 && ! blocking_error (errno))
                {
                    error (1, errno, "flow control read failed");
                }
            }
2980    }
}
#endif /* SERVER_FLOWCONTROL */

void server_output_not_carried_for_file (struct file_info* finfo, Vers_TS* vers)
{
  char* server = NULL;
  char* path = NULL;

  server = strchr (vers->vn_remote, ':');
2990  if (server != NULL) {
    server = strchr (server + 1, ':');
  }
  if (server != NULL) {
    server += 1;
    path = strchr (server, ':');
  }
  if (path != NULL) {
    *path = '\0';
    path++;
3000  }

  if ((server == NULL) || (path == NULL)) {
    error (0, 0, "Remote entry invalid");
  } else {
    /* Figure out the path on the other server where the file is located
     * Replace our root with remote root and remove filename component
     */

    char* remote_path = xmalloc (strlen (vers -> srcfile -> path) + strlen (path) + 1);
3010    sprintf (remote_path, "%s%s", path, vers->srcfile->path + strlen (CVSroot_directory));
    *strrchr (remote_path, '/') = '\0';

    if (!server_active) {
      printf ("Not-carried %s %s\n%s\n%s\n%s\n", finfo -> fullname, vers->tag, server, path, remote_path);
    } else {
      server_output_not_carried (finfo -> fullname, vers->tag, server, path, remote_path);
    }

    free (remote_path);
3020  }
}

void server_output_not_carried (char* file, char* rev, char* server, char* root, char* repository)
{
  buf_output0 (protocol, "Not-carried ");
  buf_output0 (protocol, file);
  buf_output0 (protocol, " ");
  buf_output0 (protocol, rev);
  buf_output0 (protocol, "\n");
3030  buf_output0 (protocol, server);
  buf_output0 (protocol, "\n");
  buf_output0 (protocol, root);
  buf_output0 (protocol, "\n");
  buf_output0 (protocol, repository);
  buf_output0 (protocol, "\n");
  buf_send_counted (protocol);
}

/* This variable commented in server.h.  */
3040 char *server_dir = NULL;

static void output_dir PROTO((char *, char *));

static void
output_dir (update_dir, repository)
    char *update_dir;
    char *repository;
{
    if (server_dir != NULL)
3050    {
        buf_output0 (protocol, server_dir);
        buf_output0 (protocol, "/");
    }
    if (update_dir[0] == '\0')
        buf_output0 (protocol, ".");
    else
        buf_output0 (protocol, update_dir);
    buf_output0 (protocol, "/\n");
```

```
    buf_output0 (protocol, repository);
    buf_output0 (protocol, "/");
}

/*
 * Entries line that we are squirreling away to send to the client when
 * we are ready.
 */
static char *entries_line;

/*
 * File which has been Scratch_File'd, we are squirreling away that fact
 * to inform the client when we are ready.
 */
static char *scratched_file;

/*
 * The scratched_file will need to be removed as well as having its entry
 * removed.
 */
static int kill_scratched_file;

void
server_register (name, version, timestamp, options, tag, date, conflict, repository)
    char *name;
    char *version;
    char *timestamp;
    char *options;
    char *tag;
    char *date;
    char *conflict;
    char *repository;
{
    int len;

    if (options == NULL)
        options = "";

    if (trace)
    {
        (void) fprintf (stderr,
                        "%c-> server_register(%s, %s, %s, %s, %s, %s, %s, %s)\n",
                        (server_active) ? 'S' : ' ', /* silly */
                        name, version, timestamp ? timestamp : "", options,
                        tag ? tag : "", date ? date : "",
                        conflict ? conflict : "", repository);
    }

    if (entries_line != NULL)
    {
        /*
         * If CVS decides to Register it more than once (which happens
         * on "cvs update foo/foo.c" where foo and foo.c are already
         * checked out), use the last of the entries lines Register'd.
         */
        free (entries_line);
    }

    /*
     * I have reports of Scratch_Entry and Register both happening, in
     * two different cases.  Using the last one which happens is almost
     * surely correct; I haven't tracked down why they both happen (or
     * even verified that they are for the same file).
     */
    if (scratched_file != NULL)
    {
        free (scratched_file);
        scratched_file = NULL;
    }

    len = (strlen (name) + strlen (version) + strlen (options) + 80);
    if (tag)
        len += strlen (tag);
    if (date)
        len += strlen (date);
        if (repository)
        len += strlen (repository) + 1;

    entries_line = xmalloc (len);
    sprintf (entries_line, "/%s/%s/", name, version);
    if (conflict != NULL)
    {
        strcat (entries_line, "+=");
    }
    strcat (entries_line, "/");
    strcat (entries_line, options);
    strcat (entries_line, "/");
    if (tag != NULL)
    {
        strcat (entries_line, "T");
```

```
        strcat (entries_line, tag);
    }
    else if (date != NULL)
    {
        strcat (entries_line, "D");
        strcat (entries_line, date);
    }
    strcat (entries_line, "/");
    if (repository != NULL) {
      strcat (entries_line, repository);
    }
}

void
server_scratch (fname)
    char *fname;
{
    /*
     * I have reports of Scratch_Entry and Register both happening, in
     * two different cases.  Using the last one which happens is almost
     * surely correct; I haven't tracked down why they both happen (or
     * even verified that they are for the same file).
     */
    if (entries_line != NULL)
    {
        free (entries_line);
        entries_line = NULL;
    }

    if (scratched_file != NULL)
    {
        buf_output0 (protocol,
                     "E CVS server internal error: duplicate Scratch_Entry\n");
        buf_send_counted (protocol);
        return;
    }
    scratched_file = xstrdup (fname);
    kill_scratched_file = 1;
}

void
server_scratch_entry_only ()
{
    kill_scratched_file = 0;
}

/* Print a new entries line, from a previous server_register.  */
static void
new_entries_line ()
{
    if (entries_line)
    {
        buf_output0 (protocol, entries_line);
        buf_output (protocol, "\n", 1);
    }
    else
        /* Return the error message as the Entries line.  */
        buf_output0 (protocol,
                     "CVS server internal error: Register missing\n");
    free (entries_line);
    entries_line = NULL;
}


static void
serve_ci (arg)
    char *arg;
{
    do_cvs_command ("commit", commit);
}

static void
checked_in_response (file, update_dir, repository)
    char *file;
    char *update_dir;
    char *repository;
{
    if (supported_response ("Mode"))
    {
        struct stat sb;
        char *mode_string;

        if ( CVS_STAT (file, &sb) < 0)
        {
            /* Not clear to me why the file would fail to exist, but it
               was happening somewhere in the testsuite.  */
            if (!existence_error (errno))
                error (0, errno, "cannot stat %s", file);
        }
        else
```

```
        {
3240        buf_output0 (protocol, "Mode ");
            mode_string = mode_to_string (sb.st_mode);
            buf_output0 (protocol, mode_string);
            buf_output0 (protocol, "\n");
            free (mode_string);
        }
    }

    buf_output0 (protocol, "Checked-in ");
    output_dir (update_dir, repository);
3250 buf_output0 (protocol, file);
    buf_output (protocol, "\n", 1);
    new_entries_line ();
}

void
server_checked_in (file, update_dir, repository)
    char *file;
    char *update_dir;
    char *repository;
3260 {
    if (noexec)
        return;
    if (scratched_file != NULL && entries_line == NULL)
    {
        /*
         * This happens if we are now doing a "cvs remove" after a previous
         * "cvs add" (without a "cvs ci" in between).
         */
        buf_output0 (protocol, "Remove-entry ");
3270    output_dir (update_dir, repository);
        buf_output0 (protocol, file);
        buf_output (protocol, "\n", 1);
        free (scratched_file);
        scratched_file = NULL;
    }
    else
    {
        checked_in_response (file, update_dir, repository);
    }
3280 buf_send_counted (protocol);
}

void
server_update_entries (file, update_dir, repository, updated)
    char *file;
    char *update_dir;
    char *repository;
    enum server_updated_arg4 updated;
{
3290 if (noexec)
        return;
    if (updated == SERVER_UPDATED)
        checked_in_response (file, update_dir, repository);
    else
    {
        if (!supported_response ("New-entry"))
            return;
        buf_output0 (protocol, "New-entry ");
        output_dir (update_dir, repository);
3300    buf_output0 (protocol, file);
        buf_output (protocol, "\n", 1);
        new_entries_line ();
    }

    buf_send_counted (protocol);
}

static void
serve_update (arg)
3310 char *arg;
{
    do_cvs_command ("update", update);
}

static void
serve_diff (arg)
    char *arg;
{
    do_cvs_command ("diff", diff);
3320 }

static void
serve_log (arg)
    char *arg;
{
    do_cvs_command ("log", cvslog);
}
```

```
static void
serve_add (arg)
    char *arg;
{
    do_cvs_command ("add", add);
}

static void
serve_remove (arg)
    char *arg;
{
    do_cvs_command ("remove", cvsremove);
}

static void
serve_status (arg)
    char *arg;
{
    do_cvs_command ("status", cvsstatus);
}

static void
serve_rdiff (arg)
    char *arg;
{
    do_cvs_command ("rdiff", patch);
}

static void
serve_tag (arg)
    char *arg;
{
    do_cvs_command ("cvstag", cvstag);
}

static void
serve_rtag (arg)
    char *arg;
{
    do_cvs_command ("rtag", rtag);
}

static void
serve_import (arg)
    char *arg;
{
    do_cvs_command ("import", import);
}

static void
serve_admin (arg)
    char *arg;
{
    do_cvs_command ("admin", admin);
}

static void
serve_history (arg)
    char *arg;
{
    do_cvs_command ("history", history);
}

static void
serve_release (arg)
    char *arg;
{
    do_cvs_command ("release", release);
}

static void serve_watch_on PROTO ((char *));

static void
serve_watch_on (arg)
    char *arg;
{
    do_cvs_command ("watch_on", watch_on);
}

static void serve_watch_off PROTO ((char *));

static void
serve_watch_off (arg)
    char *arg;
{
    do_cvs_command ("watch_off", watch_off);
}

static void serve_watch_add PROTO ((char *));
```

```
static void
serve_watch_add (arg)
    char *arg;
{
    do_cvs_command ("watch_add", watch_add);
}

static void serve_watch_remove PROTO ((char *));

static void
serve_watch_remove (arg)
    char *arg;
{
    do_cvs_command ("watch_remove", watch_remove);
}

static void serve_watchers PROTO ((char *));

static void
serve_watchers (arg)
    char *arg;
{
    do_cvs_command ("watchers", watchers);
}

static void serve_editors PROTO ((char *));

static void
serve_editors (arg)
    char *arg;
{
    do_cvs_command ("editors", editors);
}

static int noop PROTO ((int, char **));

static int
noop (argc, argv)
    int argc;
    char **argv;
{
    return 0;
}

static void serve_noop PROTO ((char *));

static void
serve_noop (arg)
    char *arg;
{
    do_cvs_command ("noop", noop);
}

static void serve_init PROTO ((char *));

static void
serve_init (arg)
    char *arg;
{
    if (!isabsolute (arg))
    {
        if (alloc_pending (80 + strlen (arg)))
            sprintf (pending_error_text,
                     "E Root %s must be an absolute pathname", arg);
        /* Fall through to do_cvs_command which will return the
           actual error.  */
    }
    set_local_cvsroot (arg);

    do_cvs_command ("init", init);
}

static void serve_annotate PROTO ((char *));

static void
serve_annotate (arg)
    char *arg;
{
    do_cvs_command ("annotate", annotate);
}

static void
serve_co (arg)
    char *arg;
{
    char *tempdir;
    int status;

    if (print_pending_error ())
        return;
```

```
    if (!isdir (CVSADM))
    {
        /*
         * The client has not sent a "Repository" line.  Check out
         * into a pristine directory.
         */
        tempdir = malloc (strlen (server_temp_dir) + 80);
        if (tempdir == NULL)
        {
            buf_output0 (buf_to_net, "E Out of memory\n");
            return;
        }
        strcpy (tempdir, server_temp_dir);
        strcat (tempdir, "/checkout-dir");
        status = mkdir_p (tempdir);
        if (status != 0 && status != EEXIST)
        {
            buf_output0 (buf_to_net, "E Cannot create ");
            buf_output0 (buf_to_net, tempdir);
            buf_append_char (buf_to_net, '\n');
            print_error (errno);
            free (tempdir);
            return;
        }

        if ( CVS_CHDIR (tempdir) < 0)
        {
            buf_output0 (buf_to_net, "E Cannot change to directory ");
            buf_output0 (buf_to_net, tempdir);
            buf_append_char (buf_to_net, '\n');
            print_error (errno);
            free (tempdir);
            return;
        }
        free (tempdir);
    }

    /* Compensate for server_export()'s setting of command_name.
     *
     * [It probably doesn't matter if do_cvs_command() gets "export"
     *  or "checkout", but we ought to be accurate where possible.]
     */
    do_cvs_command ((strcmp (command_name, "export") == 0) ?
                    "export" : "checkout",
                    checkout);
}

static void
serve_export (arg)
    char *arg;
{
    /* Tell checkout() to behave like export not checkout.  */
    command_name = "export";
    serve_co (arg);
}

void
server_copy_file (file, update_dir, repository, newfile)
    char *file;
    char *update_dir;
    char *repository;
    char *newfile;
{
    if (!supported_response ("Copy-file"))
        return;
    buf_output0 (protocol, "Copy-file ");
    output_dir (update_dir, repository);
    buf_output0 (protocol, file);
    buf_output0 (protocol, "\n");
    buf_output0 (protocol, newfile);
    buf_output0 (protocol, "\n");
}

/* See server.h for description.  */

void
server_modtime (finfo, vers_ts)
    struct file_info *finfo;
    Vers_TS *vers_ts;
{
    char date[MAXDATELEN];
    int year, month, day, hour, minute, second;
    /* Note that these strings are specified in RFC822 and do not vary
       according to locale.  */
    static const char *const month_names[] =
      {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    assert (vers_ts->vn_rcs != NULL);
```

```
    if (!supported_response ("Mod-time"))
        return;

    /* The only hard part about this routine is converting the date
       formats.  In terms of functionality it all boils down to the
       call to RCS_getrevtime.  */
    if (RCS_getrevtime (finfo->rcs, vers_ts->vn_rcs, date, 0) == (time_t) -1)
        /* FIXME? should we be printing some kind of warning?  For one
           thing I'm not 100% sure whether this happens in non-error
           circumstances.  */
        return;

    sscanf (date, SDATEFORM, &year, &month, &day, &hour, &minute, &second);
    sprintf (date, "%d %s %d %d:%d:%d -0000", day,
             month < 1 || month > 12 ? "???" : month_names[month - 1],
             year, hour, minute, second);
    buf_output0 (protocol, "Mod-time ");
    buf_output0 (protocol, date);
    buf_output0 (protocol, "\n");
}

/* See server.h for description.  */

#if defined (USE_PROTOTYPES) ? USE_PROTOTYPES : defined (__STDC__)
/* Need to prototype because mode_t might be smaller than int.  */
void
server_updated (
    struct file_info *finfo,
    Vers_TS *vers,
    enum server_updated_arg4 updated,
    mode_t mode,
    unsigned char *checksum,
    struct buffer *filebuf)
#else
void
server_updated (finfo, vers, updated, mode, checksum, filebuf)
    struct file_info *finfo;
    Vers_TS *vers;
    enum server_updated_arg4 updated;
    mode_t mode;
    unsigned char *checksum;
    struct buffer *filebuf;
#endif
{
    if (noexec)
    {
        /* Hmm, maybe if we did the same thing for entries_file, we
           could get rid of the kludges in server_register and
           server_scratch which refrain from warning if both
           Scratch_Entry and Register get called.  Maybe.  */
        if (scratched_file)
        {
            free (scratched_file);
            scratched_file = NULL;
        }
        return;
    }

    if (entries_line != NULL && scratched_file == NULL)
    {
        FILE *f;
        struct buffer_data *list, *last;
        unsigned long size;
        char size_text[80];

        if (filebuf != NULL)
        {
            size = buf_length (filebuf);
            if (mode == (mode_t) -1)
                error (1, 0, "\
CVS server internal error: no mode in server_updated");
        }
        else
        {
            struct stat sb;

            if ( CVS_STAT (finfo->file, &sb) < 0)
            {
                if (existence_error (errno))
                {
                    /* If we have a sticky tag for a branch on which
                       the file is dead, and cvs update the directory,
                       it gets a T_CHECKOUT but no file.  So in this
                       case just forget the whole thing.  */
                    free (entries_line);
                    entries_line = NULL;
                    goto done;
                }
                error (1, errno, "reading %s", finfo->fullname);
```

```
            }
3690        size = sb.st_size;
            if (mode == (mode_t) -1)
            {
                /* FIXME: When we check out files the umask of the
                   server (set in .bashrc if rsh is in use) affects
                   what mode we send, and it shouldn't.  */
                mode = sb.st_mode;
            }
        }

3700    if (checksum != NULL)
        {
            static int checksum_supported = -1;

            if (checksum_supported == -1)
            {
                checksum_supported = supported_response ("Checksum");
            }

            if (checksum_supported)
3710        {
                int i;
                char buf[3];

                buf_output0 (protocol, "Checksum ");
                for (i = 0; i < 16; i++)
                {
                    sprintf (buf, "%02x", (unsigned int) checksum[i]);
                    buf_output0 (protocol, buf);
                }
3720            buf_append_char (protocol, '\n');
            }
        }

        if (updated == SERVER_UPDATED)
        {
            Node *node;
            Entnode *entnode;

            if (!(supported_response ("Created")
3730              && supported_response ("Update-existing")))
                buf_output0 (protocol, "Updated ");
            else
            {
                assert (vers != NULL);
                if (vers->ts_user == NULL)
                    buf_output0 (protocol, "Created ");
                else
                    buf_output0 (protocol, "Update-existing ");
            }
3740
            /* Now munge the entries to say that the file is unmodified,
               in case we end up processing it again (e.g. modules3-6
               in the testsuite).  */
            node = findnode_fn (finfo->entries, finfo->file);
            entnode = (Entnode *)node->data;
            free (entnode->timestamp);
            entnode->timestamp = xstrdup ("=");
        }
        else if (updated == SERVER_MERGED)
3750        buf_output0 (protocol, "Merged ");
        else if (updated == SERVER_PATCHED)
            buf_output0 (protocol, "Patched ");
        else if (updated == SERVER_RCS_DIFF)
            buf_output0 (protocol, "Rcs-diff ");
        else
            abort ();
        output_dir (finfo->update_dir, finfo->repository);
        buf_output0 (protocol, finfo->file);
        buf_output (protocol, "\n", 1);
3760
        new_entries_line ();

        {
            char *mode_string;

            mode_string = mode_to_string (mode);
            buf_output0 (protocol, mode_string);
            buf_output0 (protocol, "\n");
            free (mode_string);
3770    }

        list = last = NULL;
        if (size > 0)
        {
            /* Throughout this section we use binary mode to read the
               file we are sending.  The client handles any line ending
               translation if necessary.  */
```

```
            if (file_gzip_level
                /*
                 * For really tiny files, the gzip process startup
                 * time will outweigh the compression savings.  This
                 * might be computable somehow; using 100 here is just
                 * a first approximation.
                 */
                && size > 100)
            {
                int status, fd, gzip_status;
                pid_t gzip_pid;

                /* Callers must avoid passing us a buffer if
                   file_gzip_level is set.  We could handle this case,
                   but it's not worth it since this case never arises
                   with a current client and server.  */
                if (filebuf != NULL)
                    error (1, 0, "\
CVS server internal error: unhandled case in server_updated");

                fd = CVS_OPEN (finfo->file, O_RDONLY | OPEN_BINARY, 0);
                if (fd < 0)
                    error (1, errno, "reading %s", finfo->fullname);
                fd = filter_through_gzip (fd, 1, file_gzip_level, &gzip_pid);
                f = fdopen (fd, "rb");
                status = buf_read_file_to_eof (f, &list, &last);
                size = buf_chain_length (list);
                if (status == -2)
                    (*protocol->memory_error) (protocol);
                else if (status != 0)
                    error (1, ferror (f) ? errno : 0, "reading %s",
                           finfo->fullname);
                if (fclose (f) == EOF)
                    error (1, errno, "reading %s", finfo->fullname);
                if (waitpid (gzip_pid, &gzip_status, 0) == -1)
                    error (1, errno, "waiting for gzip process %ld",
                           (long) gzip_pid);
                else if (gzip_status != 0)
                    error (1, 0, "gzip exited %d", gzip_status);
                /* Prepending length with "z" is flag for using gzip here.  */
                buf_output0 (protocol, "z");
            }
            else if (filebuf == NULL)
            {
                long status;

                f = CVS_FOPEN (finfo->file, "rb");
                if (f == NULL)
                    error (1, errno, "reading %s", finfo->fullname);
                status = buf_read_file (f, size, &list, &last);
                if (status == -2)
                    (*protocol->memory_error) (protocol);
                else if (status != 0)
                    error (1, ferror (f) ? errno : 0, "reading %s",
                           finfo->fullname);
                if (fclose (f) == EOF)
                    error (1, errno, "reading %s", finfo->fullname);
            }
        }

        sprintf (size_text, "%lu\n", size);
        buf_output0 (protocol, size_text);

        if (filebuf == NULL)
            buf_append_data (protocol, list, last);
        else
        {
            buf_append_buffer (protocol, filebuf);
            buf_free (filebuf);
        }
        /* Note we only send a newline here if the file ended with one.  */

        /*
         * Avoid using up too much disk space for temporary files.
         * A file which does not exist indicates that the file is up-to-date,
         * which is now the case.  If this is SERVER_MERGED, the file is
         * not up-to-date, and we indicate that by leaving the file there.
         * I'm thinking of cases like "cvs update foo/foo.c foo".
         */
        if ((updated == SERVER_UPDATED
             || updated == SERVER_PATCHED
             || updated == SERVER_RCS_DIFF)
            && filebuf == NULL
            /* But if we are joining, we'll need the file when we call
               join_file.  */
            && !joining ())
            CVS_UNLINK (finfo->file);
    }
    else if (scratched_file != NULL && entries_line == NULL)
    {
```

```
        if (strcmp (scratched_file, finfo->file) != 0)
            error (1, 0,
                   "CVS server internal error: `%s' vs. `%s' scratched",
                   scratched_file,
                   finfo->file);
        free (scratched_file);
        scratched_file = NULL;

        if (kill_scratched_file)
            buf_output0 (protocol, "Removed ");
        else
            buf_output0 (protocol, "Remove-entry ");
        output_dir (finfo->update_dir, finfo->repository);
        buf_output0 (protocol, finfo->file);
        buf_output (protocol, "\n", 1);
    }
    else if (scratched_file == NULL && entries_line == NULL)
    {
        /*
         * This can happen with death support if we were processing
         * a dead file in a checkout.
         */
    }
    else
        error (1, 0,
               "CVS server internal error: Register *and* Scratch_Entry.\n");
    buf_send_counted (protocol);
  done:;
}

/* Return whether we should send patches in RCS format.  */

int
server_use_rcs_diff ()
{
    return supported_response ("Rcs-diff");
}

void
server_set_entstat (update_dir, repository)
    char *update_dir;
    char *repository;
{
    static int set_static_supported = -1;
    if (set_static_supported == -1)
        set_static_supported = supported_response ("Set-static-directory");
    if (!set_static_supported) return;

    buf_output0 (protocol, "Set-static-directory ");
    output_dir (update_dir, repository);
    buf_output0 (protocol, "\n");
    buf_send_counted (protocol);
}

void
server_clear_entstat (update_dir, repository)
     char *update_dir;
     char *repository;
{
    static int clear_static_supported = -1;
    if (clear_static_supported == -1)
        clear_static_supported = supported_response ("Clear-static-directory");
    if (!clear_static_supported) return;

    if (noexec)
        return;

    buf_output0 (protocol, "Clear-static-directory ");
    output_dir (update_dir, repository);
    buf_output0 (protocol, "\n");
    buf_send_counted (protocol);
}

void
server_set_sticky (update_dir, repository, tag, date, nonbranch)
    char *update_dir;
    char *repository;
    char *tag;
    char *date;
    int nonbranch;
{
    static int set_sticky_supported = -1;

    assert (update_dir != NULL);

    if (set_sticky_supported == -1)
        set_sticky_supported = supported_response ("Set-sticky");
    if (!set_sticky_supported) return;

    if (noexec)
```

```
        return;

    if (tag == NULL && date == NULL)
    {
        buf_output0 (protocol, "Clear-sticky ");
        output_dir (update_dir, repository);
        buf_output0 (protocol, "\n");
    }
    else
    {
        buf_output0 (protocol, "Set-sticky ");
        output_dir (update_dir, repository);
        buf_output0 (protocol, "\n");
        if (tag != NULL)
        {
            if (nonbranch)
                buf_output0 (protocol, "N");
            else
                buf_output0 (protocol, "T");
            buf_output0 (protocol, tag);
        }
        else
        {
            buf_output0 (protocol, "D");
            buf_output0 (protocol, date);
        }
        buf_output0 (protocol, "\n");
    }
    buf_send_counted (protocol);
}

struct template_proc_data
{
    char *update_dir;
    char *repository;
};

/* Here as a static until we get around to fixing Parse_Info to pass along
   a void * for it.  */
static struct template_proc_data *tpd;

static int
template_proc (repository, template)
    char *repository;
    char *template;
{
    FILE *fp;
    char buf[1024];
    size_t n;
    struct stat sb;
    struct template_proc_data *data = tpd;

    if (!supported_response ("Template"))
        /* Might want to warn the user that the rcsinfo feature won't work.  */
        return 0;
    buf_output0 (protocol, "Template ");
    output_dir (data->update_dir, data->repository);
    buf_output0 (protocol, "\n");

    fp = CVS_FOPEN (template, "rb");
    if (fp == NULL)
    {
        error (0, errno, "Couldn't open rcsinfo template file %s", template);
        return 1;
    }
    if (fstat (fileno (fp), &sb) < 0)
    {
        error (0, errno, "cannot stat rcsinfo template file %s", template);
        return 1;
    }
    sprintf (buf, "%ld\n", (long) sb.st_size);
    buf_output0 (protocol, buf);
    while (!feof (fp))
    {
        n = fread (buf, 1, sizeof buf, fp);
        buf_output (protocol, buf, n);
        if (ferror (fp))
        {
            error (0, errno, "cannot read rcsinfo template file %s", template);
            (void) fclose (fp);
            return 1;
        }
    }
    if (fclose (fp) < 0)
        error (0, errno, "cannot close rcsinfo template file %s", template);
    return 0;
}

void
server_template (update_dir, repository)
```

```
    char *update_dir;
    char *repository;
{
    struct template_proc_data data;
    data.update_dir = update_dir;
    data.repository = repository;
    tpd = &data;
    (void) Parse_Info (CVSROOTADM_RCSINFO, repository, template_proc, 1);
}

static void
serve_gzip_contents (arg)
     char *arg;
{
    int level;
    level = atoi (arg);
    if (level == 0)
        level = 6;
    file_gzip_level = level;
}

static void
serve_gzip_stream (arg)
     char *arg;
{
    int level;
    level = atoi (arg);
    if (level == 0)
        level = 6;

    /* All further communication with the client will be compressed.  */

    buf_to_net = compress_buffer_initialize (buf_to_net, 0, level,
                                             buf_to_net->memory_error);
    buf_from_net = compress_buffer_initialize (buf_from_net, 1, level,
                                               buf_from_net->memory_error);
}

/* Tell the client about RCS options set in CVSROOT/cvswrappers. */
static void
serve_wrapper_sendme_rcs_options (arg)
     char *arg;
{
    /* Actually, this is kind of sdrawkcab-ssa: the client wants
     * verbatim lines from a cvswrappers file, but the server has
     * already parsed the cvswrappers file into the wrap_list struct.
     * Therefore, the server loops over wrap_list, unparsing each
     * entry before sending it.
     */
    char *wrapper_line = NULL;

    wrap_setup ();

    for (wrap_unparse_rcs_options (&wrapper_line, 1);
         wrapper_line;
         wrap_unparse_rcs_options (&wrapper_line, 0))
    {
        buf_output0 (buf_to_net, "Wrapper-rcsOption ");
        buf_output0 (buf_to_net, wrapper_line);
        buf_output0 (buf_to_net, "\012");;
        free (wrapper_line);
    }

    buf_output0 (buf_to_net, "ok\012");

    /* The client is waiting for us, so we better send the data now.  */
    buf_flush (buf_to_net, 1);
}


static void
serve_ignore (arg)
    char *arg;
{
    /*
     * Just ignore this command.  This is used to support the
     * update-patches command, which is not a real command, but a signal
     * to the client that update will accept the -u argument.
     */
}

static int
expand_proc (pargc, argv, where, mwhere, mfile, shorten,
             local_specified, omodule, msg)
    int *pargc;
    char **argv;
    char *where;
    char *mwhere;
    char *mfile;
    int shorten;
```

```
    int local_specified;
4140 char *omodule;
    char *msg;
{
    int i;
    char *dir = argv[0];

    /* If mwhere has been specified, the thing we're expanding is a
       module – just return its name so the client will ask for the
       right thing later.  If it is an alias or a real directory,
       mwhere will not be set, so send out the appropriate
4150   expansion. */

    if (mwhere != NULL)
    {
        buf_output0 (buf_to_net, "Module-expansion ");
        if (server_dir != NULL)
        {
            buf_output0 (buf_to_net, server_dir);
            buf_output0 (buf_to_net, "/");
        }
4160    buf_output0 (buf_to_net, mwhere);
        if (mfile != NULL)
        {
            buf_append_char (buf_to_net, '/');
            buf_output0 (buf_to_net, mfile);
        }
        buf_append_char (buf_to_net, '\n');
    }
    else
    {
4170    /* We may not need to do this anymore – check the definition
           of aliases before removing */
        if (*pargc == 1)
        {
            buf_output0 (buf_to_net, "Module-expansion ");
            if (server_dir != NULL)
            {
                buf_output0 (buf_to_net, server_dir);
                buf_output0 (buf_to_net, "/");
            }
4180        buf_output0 (buf_to_net, dir);
            buf_append_char (buf_to_net, '\n');
        }
        else
        {
            for (i = 1; i < *pargc; ++i)
            {
                buf_output0 (buf_to_net, "Module-expansion ");
                if (server_dir != NULL)
                {
4190                buf_output0 (buf_to_net, server_dir);
                    buf_output0 (buf_to_net, "/");
                }
                buf_output0 (buf_to_net, dir);
                buf_append_char (buf_to_net, '/');
                buf_output0 (buf_to_net, argv[i]);
                buf_append_char (buf_to_net, '\n');
            }
        }
    }
4200 return 0;
}

static void
serve_expand_modules (arg)
    char *arg;
{
    int i;
    int err;
    DBM *db;
4210 err = 0;

    server_expanding = 1;
    db = open_module ();
    for (i = 1; i < argument_count; i++)
        err += do_module (db, argument_vector[i],
                          CHECKOUT, "Updating", expand_proc,
                          NULL, 0, 0, 0,
                          (char *) NULL);
    close_module (db);
4220 server_expanding = 0;
    {
        /* argument_vector[0] is a dummy argument, we don't mess with it.  */
        char **cp;
        for (cp = argument_vector + 1;
             cp < argument_vector + argument_count;
             ++cp)
            free (*cp);
```

```
        argument_count = 1;
    }
    if (err)
        /* We will have printed an error message already.  */
        buf_output0 (buf_to_net, "error  \n");
    else
        buf_output0 (buf_to_net, "ok\n");

    /* The client is waiting for the module expansions, so we must
       send the output now.  */
    buf_flush (buf_to_net, 1);
}

void
server_prog (dir, name, which)
    char *dir;
    char *name;
    enum progs which;
{
    if (!supported_response ("Set-checkin-prog"))
    {
        buf_output0 (buf_to_net, "E \
warning: this client does not support -i or -u flags in the modules file.\n");
        return;
    }
    switch (which)
    {
        case PROG_CHECKIN:
            buf_output0 (buf_to_net, "Set-checkin-prog ");
            break;
        case PROG_UPDATE:
            buf_output0 (buf_to_net, "Set-update-prog ");
            break;
    }
    buf_output0 (buf_to_net, dir);
    buf_append_char (buf_to_net, '\n');
    buf_output0 (buf_to_net, name);
    buf_append_char (buf_to_net, '\n');
}

static void
serve_checkin_prog (arg)
    char *arg;
{
    FILE *f;
    f = CVS_FOPEN (CVSADM_CIPROG, "w+");
    if (f == NULL)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
            sprintf (pending_error_text, "E cannot open %s", CVSADM_CIPROG);
        return;
    }
    if (fprintf (f, "%s\n", arg) < 0)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
            sprintf (pending_error_text,
                     "E cannot write to %s", CVSADM_CIPROG);
        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_CIPROG)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_CIPROG);
        return;
    }
}

static void
serve_update_prog (arg)
    char *arg;
{
    FILE *f;

    /* Before we do anything we need to make sure we are not in readonly
       mode.  */
    if (!check_command_legal_p ("commit"))
    {
        /* I might be willing to make this a warning, except we lack the
           machinery to do so.  */
        if (alloc_pending (80))
            sprintf (pending_error_text, "\
E Flag -u in modules not allowed in readonly mode");
        return;
    }

    f = CVS_FOPEN (CVSADM_UPROG, "w+");
    if (f == NULL)
```

```
    {
4320        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_UPROG)))
            sprintf (pending_error_text, "E cannot open %s", CVSADM_UPROG);
        return;
    }
    if (fprintf (f, "%s\n", arg) < 0)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_UPROG)))
            sprintf (pending_error_text, "E cannot write to %s", CVSADM_UPROG);
4330        return;
    }
    if (fclose (f) == EOF)
    {
        pending_error = errno;
        if (alloc_pending (80 + strlen (CVSADM_UPROG)))
            sprintf (pending_error_text, "E cannot close %s", CVSADM_UPROG);
        return;
    }
}
4340
static void serve_valid_requests PROTO((char *arg));

#endif /* SERVER_SUPPORT */
#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)

/*
 * Parts of this table are shared with the client code,
 * but the client doesn't need to know about the handler
 * functions.
4350 */

struct request requests[] =
{
#ifdef SERVER_SUPPORT
#define REQ_LINE(n, f, s) {n, f, s}
#else
#define REQ_LINE(n, f, s) {n, s}
#endif

4360  REQ_LINE("Root", serve_root, rq_essential),
  REQ_LINE("Valid-responses", serve_valid_responses, rq_essential),
  REQ_LINE("valid-requests", serve_valid_requests, rq_essential),
  REQ_LINE("Repository", serve_repository, rq_optional),
  REQ_LINE("Directory", serve_directory, rq_essential),
  REQ_LINE("Max-dotdot", serve_max_dotdot, rq_optional),
  REQ_LINE("Static-directory", serve_static_directory, rq_optional),
  REQ_LINE("Sticky", serve_sticky, rq_optional),
  REQ_LINE("Checkin-prog", serve_checkin_prog, rq_optional),
  REQ_LINE("Update-prog", serve_update_prog, rq_optional),
4370  REQ_LINE("Entry", serve_entry, rq_essential),
  REQ_LINE("Kopt", serve_kopt, rq_optional),
  REQ_LINE("Modified", serve_modified, rq_essential),
  REQ_LINE("Remote-revision", serve_remote_revision, rq_optional),
  REQ_LINE("Is-modified", serve_is_modified, rq_optional),

  /* The client must send this request to interoperate with CVS 1.5
     through 1.9 servers.  The server must support it (although it can
     be and is a noop) to interoperate with CVS 1.5 to 1.9 clients.  */
  REQ_LINE("UseUnchanged", serve_enable_unchanged, rq_enableme),
4380
  REQ_LINE("Unchanged", serve_unchanged, rq_essential),
  REQ_LINE("Notify", serve_notify, rq_optional),
  REQ_LINE("Questionable", serve_questionable, rq_optional),
  REQ_LINE("Case", serve_case, rq_optional),
  REQ_LINE("Argument", serve_argument, rq_essential),
  REQ_LINE("Argumentx", serve_argumentx, rq_essential),
  REQ_LINE("Global_option", serve_global_option, rq_optional),
  REQ_LINE("Gzip-stream", serve_gzip_stream, rq_optional),
  REQ_LINE("wrapper-sendme-rcsOptions",
4390         serve_wrapper_sendme_rcs_options,
         rq_optional),
  REQ_LINE("Set", serve_set, rq_optional),
#ifdef ENCRYPTION
#  ifdef HAVE_KERBEROS
  REQ_LINE("Kerberos-encrypt", serve_kerberos_encrypt, rq_optional),
#  endif
#  ifdef HAVE_GSSAPI
  REQ_LINE("Gssapi-encrypt", serve_gssapi_encrypt, rq_optional),
#  endif
4400 #endif
#ifdef HAVE_GSSAPI
  REQ_LINE("Gssapi-authenticate", serve_gssapi_authenticate, rq_optional),
#endif
  REQ_LINE("expand-modules", serve_expand_modules, rq_optional),
  REQ_LINE("ci", serve_ci, rq_essential),
  REQ_LINE("co", serve_co, rq_essential),
  REQ_LINE("update", serve_update, rq_essential),
  REQ_LINE("diff", serve_diff, rq_optional),
```

```
    REQ_LINE("log", serve_log, rq_optional),
    REQ_LINE("add", serve_add, rq_optional),
    REQ_LINE("remove", serve_remove, rq_optional),
    REQ_LINE("update-patches", serve_ignore, rq_optional),
    REQ_LINE("gzip-file-contents", serve_gzip_contents, rq_optional),
    REQ_LINE("status", serve_status, rq_optional),
    REQ_LINE("rdiff", serve_rdiff, rq_optional),
    REQ_LINE("tag", serve_tag, rq_optional),
    REQ_LINE("rtag", serve_rtag, rq_optional),
    REQ_LINE("import", serve_import, rq_optional),
    REQ_LINE("admin", serve_admin, rq_optional),
    REQ_LINE("export", serve_export, rq_optional),
    REQ_LINE("history", serve_history, rq_optional),
    REQ_LINE("release", serve_release, rq_optional),
    REQ_LINE("watch-on", serve_watch_on, rq_optional),
    REQ_LINE("watch-off", serve_watch_off, rq_optional),
    REQ_LINE("watch-add", serve_watch_add, rq_optional),
    REQ_LINE("watch-remove", serve_watch_remove, rq_optional),
    REQ_LINE("watchers", serve_watchers, rq_optional),
    REQ_LINE("editors", serve_editors, rq_optional),
    REQ_LINE("init", serve_init, rq_optional),
    REQ_LINE("annotate", serve_annotate, rq_optional),
    REQ_LINE("noop", serve_noop, rq_optional),
    REQ_LINE(NULL, NULL, rq_optional)

#undef REQ_LINE
};

#endif /* SERVER_SUPPORT or CLIENT_SUPPORT */
#ifdef SERVER_SUPPORT

static void
serve_valid_requests (arg)
     char *arg;
{
    struct request *rq;
    if (print_pending_error ())
        return;
    buf_output0 (buf_to_net, "Valid-requests");
    for (rq = requests; rq->name != NULL; rq++)
    {
        if (rq->func != NULL)
        {
            buf_append_char (buf_to_net, ' ');
            buf_output0 (buf_to_net, rq->name);
        }
    }
    buf_output0 (buf_to_net, "\nok\n");

    /* The client is waiting for the list of valid requests, so we
       must send the output now.  */
    buf_flush (buf_to_net, 1);
}

#ifdef sun
/*
 * Delete temporary files.  SIG is the signal making this happen, or
 * 0 if not called as a result of a signal.
 */
static int command_pid_is_dead;
static void wait_sig (sig)
     int sig;
{
    int status;
    pid_t r = wait (&status);
    if (r == command_pid)
        command_pid_is_dead++;
}
#endif

void
server_cleanup (sig)
    int sig;
{
    /* Do "rm -rf" on the temp directory.  */
    int status;
    int save_noexec;

    if (buf_to_net != NULL)
    {
        /* FIXME: If this is not the final call from server, this
           could deadlock, because the client might be blocked writing
           to us.  This should not be a problem in practice, because
           we do not generate much output when the client is not
           waiting for it.  */
        set_block (buf_to_net);
        buf_flush (buf_to_net, 1);

        /* The calls to buf_shutdown are currently only meaningful
           when we are using compression.  First we shut down
```

```
            BUF_FROM_NET.  That will pick up the checksum generated
            when the client shuts down its buffer.  Then, after we have
            generated any final output, we shut down BUF_TO_NET.  */

         status = buf_shutdown (buf_from_net);
         if (status != 0)
         {
            error (0, status, "shutting down buffer from client");
            buf_flush (buf_to_net, 1);
         }
      }

      if (dont_delete_temp)
      {
         if (buf_to_net != NULL)
            (void) buf_shutdown (buf_to_net);
         return;
      }

      /* What a bogus kludge.  This disgusting code makes all kinds of
         assumptions about SunOS, and is only for a bug in that system.
         So only enable it on Suns.  */
#ifdef sun
      if (command_pid > 0)
      {
         /* To avoid crashes on SunOS due to bugs in SunOS tmpfs
            triggered by the use of rename() in RCS, wait for the
            subprocess to die.  Unfortunately, this means draining output
            while waiting for it to unblock the signal we sent it.  Yuck!  */
         int status;
         pid_t r;

         signal (SIGCHLD, wait_sig);
         if (sig)
            /* Perhaps SIGTERM would be more correct.  But the child
               process will delay the SIGINT delivery until its own
               children have exited.  */
            kill (command_pid, SIGINT);
         /* The caller may also have sent a signal to command_pid, so
            always try waiting.  First, though, check and see if it's still
            there....  */
      do_waitpid:
         r = waitpid (command_pid, &status, WNOHANG);
         if (r == 0)
            ;
         else if (r == command_pid)
            command_pid_is_dead++;
         else if (r == -1)
            switch (errno)
            {
               case ECHILD:
                  command_pid_is_dead++;
                  break;
               case EINTR:
                  goto do_waitpid;
            }
         else
            /* waitpid should always return one of the above values */
            abort ();
         while (!command_pid_is_dead)
         {
            struct timeval timeout;
            struct fd_set_wrapper readfds;
            char buf[100];
            int i;

            /* Use a non-zero timeout to avoid eating up CPU cycles.  */
            timeout.tv_sec = 2;
            timeout.tv_usec = 0;
            readfds = command_fds_to_drain;
            switch (select (max_command_fd + 1, &readfds.fds,
                            (fd_set *)0, (fd_set *)0,
                            &timeout))
            {
               case -1:
                  if (errno != EINTR)
                     abort ();
               case 0:
                  /* timeout */
                  break;
               case 1:
                  for (i = 0; i <= max_command_fd; i++)
                  {
                     if (!FD_ISSET (i, &readfds.fds))
                        continue;
                     /* this fd is non-blocking */
                     while (read (i, buf, sizeof (buf)) >= 1)
                        ;
                  }
                  break;
```

```
                default:
4590                abort ();
            }
        }
    }
#endif

    CVS_CHDIR (Tmpdir);
    /* Temporarily clear noexec, so that we clean up our temp directory
       regardless of it (this could more cleanly be handled by moving
       the noexec check to all the unlink_file_dir callers from
4600   unlink_file_dir itself). */
    save_noexec = noexec;
    noexec = 0;
    /* FIXME?  Would be nice to not ignore errors.  But what should we do?
       We could try to do this before we shut down the network connection,
       and try to notify the client (but the client might not be waiting
       for responses).  We could try something like syslog() or our own
       log file. */
    unlink_file_dir (orig_server_temp_dir);
    noexec = save_noexec;
4610
    if (buf_to_net != NULL)
        (void) buf_shutdown (buf_to_net);
}

int server_active = 0;
int server_expanding = 0;

int
server (argc, argv)
4620    int argc;
    char **argv;
{
    if (argc == -1)
    {
        static const char *const msg[] =
        {
            "Usage: %s %s\n",
            "  Normally invoked by a cvs client on a remote machine.\n",
            NULL
4630        };
        usage (msg);
    }
    /* Ignore argc and argv.  They might be from .cvsrc. */

    buf_to_net = fd_buffer_initialize (STDOUT_FILENO, 0,
                                       outbuf_memory_error);
    buf_from_net = stdio_buffer_initialize (stdin, 1, outbuf_memory_error);

    saved_output = buf_nonio_initialize (outbuf_memory_error);
4640    saved_outerr = buf_nonio_initialize (outbuf_memory_error);

    /* Since we're in the server parent process, error should use the
       protocol to report error messages. */
    error_use_protocol = 1;

    /* OK, now figure out where we stash our temporary files. */
    {
        char *p;

4650        /* The code which wants to chdir into server_temp_dir is not set
           up to deal with it being a relative path.  So give an error
           for that case. */
        if (!isabsolute (Tmpdir))
        {
            pending_error_text = malloc (80 + strlen (Tmpdir));
            if (pending_error_text == NULL)
            {
                pending_error = ENOMEM;
            }
4660        else
            {
                sprintf (pending_error_text,
                         "E Value of %s for TMPDIR is not absolute", Tmpdir);
            }
            /* FIXME: we would like this error to be persistent, that
               is, not cleared by print_pending_error.  The current client
               will exit as soon as it gets an error, but the protocol spec
               does not require a client to do so. */
        }
4670    else
        {
            int status;

            server_temp_dir = malloc (strlen (Tmpdir) + 80);
            if (server_temp_dir == NULL)
            {
                /*
                 * Strictly speaking, we're not supposed to output anything
```

```
                 * now.  But we're about to exit(), give it a try.
4680             */
                printf ("E Fatal server error, aborting.\n\
error ENOMEM Virtual memory exhausted.\n");

                /* I'm doing this manually rather than via error_exit ()
                   because I'm not sure whether we want to call server_cleanup.
                   Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
                /* Hook for OS-specific behavior, for example socket
4690               subsystems on NT and OS2 or dealing with windows
                   and arguments on Mac.  */
                SYSTEM_CLEANUP ();
#endif

                exit (EXIT_FAILURE);
            }
            strcpy (server_temp_dir, Tmpdir);

            /* Remove a trailing slash from TMPDIR if present.  */
4700        p = server_temp_dir + strlen (server_temp_dir) - 1;
            if (*p == '/')
                *p = '\0';

            /*
             * I wanted to use cvs-serv/PID, but then you have to worry about
             * the permissions on the cvs-serv directory being right.  So
             * use cvs-servPID.
             */
            strcat (server_temp_dir, "/cvs-serv");
4710
            p = server_temp_dir + strlen (server_temp_dir);
            sprintf (p, "%ld", (long) getpid ());

            orig_server_temp_dir = server_temp_dir;

            /* Create the temporary directory, and set the mode to
               700, to discourage random people from tampering with
               it.  */
            status = mkdir_p (server_temp_dir);
4720        if (status != 0 && status != EEXIST)
            {
                if (alloc_pending (80))
                    strcpy (pending_error_text,
                            "E can't create temporary directory");
                pending_error = status;
            }
#ifndef CHMOD_BROKEN
            else
            {
4730            if (chmod (server_temp_dir, S_IRWXU) < 0)
                {
                    int save_errno = errno;
                    if (alloc_pending (80))
                        strcpy (pending_error_text, "\
E cannot change permissions on temporary directory");
                    pending_error = save_errno;
                }
            }
#endif
4740    }
    }

#ifdef SIGHUP
    (void) SIG_register (SIGHUP, server_cleanup);
#endif
#ifdef SIGINT
    (void) SIG_register (SIGINT, server_cleanup);
#endif
#ifdef SIGQUIT
4750(void) SIG_register (SIGQUIT, server_cleanup);
#endif
#ifdef SIGPIPE
    (void) SIG_register (SIGPIPE, server_cleanup);
#endif
#ifdef SIGTERM
    (void) SIG_register (SIGTERM, server_cleanup);
#endif

    /* Now initialize our argument vector (for arguments from the client).  */
4760
    /* Small for testing.  */
    argument_vector_size = 1;
    argument_vector =
        (char **) malloc (argument_vector_size * sizeof (char *));
    if (argument_vector == NULL)
    {
        /*
         * Strictly speaking, we're not supposed to output anything
```

```
         * now.  But we're about to exit(), give it a try.
4770     */
        printf ("E Fatal server error, aborting.\n\
error ENOMEM Virtual memory exhausted.\n");

        /* I'm doing this manually rather than via error_exit ()
           because I'm not sure whether we want to call server_cleanup.
           Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
        /* Hook for OS-specific behavior, for example socket subsystems on
4780       NT and OS2 or dealing with windows and arguments on Mac.  */
        SYSTEM_CLEANUP ();
#endif

        exit (EXIT_FAILURE);
    }

    argument_count = 1;
    /* This gets printed if the client supports an option which the
       server doesn't, causing the server to print a usage message.
4790   FIXME: probably should be using program_name here.
       FIXME: just a nit, I suppose, but the usage message the server
       prints isn't literally true--it suggests "cvs server" followed
       by options which are for a particular command.  Might be nice to
       say something like "client apparently supports an option not supported
       by this server" or something like that instead of usage message.  */
    argument_vector[0] = "cvs server";

    while (1)
    {
4800    char *cmd, *orig_cmd;
        struct request *rq;
        int status;

        status = buf_read_line (buf_from_net, &cmd, (int *) NULL);
        if (status == -2)
        {
            buf_output0 (buf_to_net, "E Fatal server error, aborting.\n\
error ENOMEM Virtual memory exhausted.\n");
            break;
4810    }
        if (status != 0)
            break;

        orig_cmd = cmd;
        for (rq = requests; rq->name != NULL; ++rq)
            if (strncmp (cmd, rq->name, strlen (rq->name)) == 0)
            {
                int len = strlen (rq->name);
                if (cmd[len] == '\0')
4820                cmd += len;
                else if (cmd[len] == ' ')
                    cmd += len + 1;
                else
                    /*
                     * The first len characters match, but it's a different
                     * command.  e.g. the command is "cooperate" but we matched
                     * "co".
                     */
                    continue;
4830            (*rq->func) (cmd);
                break;
            }
        if (rq->name == NULL)
        {
            if (!print_pending_error ())
            {
                buf_output0 (buf_to_net, "error  unrecognized request `");
                buf_output0 (buf_to_net, cmd);
                buf_append_char (buf_to_net, '\'');
4840            buf_append_char (buf_to_net, '\n');
            }
        }
        free (orig_cmd);
    }
    server_cleanup (0);
    return 0;
}


4850 #if defined (HAVE_KERBEROS) || defined (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI)
static void switch_to_user PROTO((const char *));

static void
switch_to_user (username)
    const char *username;
{
    struct passwd *pw;
```

```
    pw = getpwnam (username);
    if (pw == NULL)
    {
        printf ("E Fatal error, aborting.\n\
error 0 %s: no such user\n", username);
        /* I'm doing this manually rather than via error_exit ()
           because I'm not sure whether we want to call server_cleanup.
           Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
        /* Hook for OS-specific behavior, for example socket subsystems on
           NT and OS2 or dealing with windows and arguments on Mac.  */
        SYSTEM_CLEANUP ();
#endif

        exit (EXIT_FAILURE);
    }

    /* FIXME?  We don't check for errors from initgroups, setuid, &c.
       I think this mainly would come up if someone is trying to run
       the server as a non-root user.  I think we should be checking for
       errors and aborting (as with the error above from getpwnam) if
       there is an error (presumably EPERM).  That means that pserver
       should continue to work right if all of the "system usernames"
       in CVSROOT/passwd match the user which the server is being run
       as (in inetd.conf), but fail otherwise.  */

#if HAVE_INITGROUPS
    initgroups (pw->pw_name, pw->pw_gid);
#endif /* HAVE_INITGROUPS */

#ifdef SETXID_SUPPORT
    /* honor the setgid bit iff set*/
    if (getgid() != getegid())
    {
        setgid (getegid ());
    }
    else
#else
    {
        setgid (pw->pw_gid);
    }
#endif

    setuid (pw->pw_uid);
    /* We don't want our umask to change file modes.  The modes should
       be set by the modes used in the repository, and by the umask of
       the client.  */
    umask (0);

#if HAVE_PUTENV
    /* Set LOGNAME and USER in the environment, in case they are
       already set to something else.  */
    {
        char *env;

        env = xmalloc (sizeof "LOGNAME=" + strlen (username));
        (void) sprintf (env, "LOGNAME=%s", username);
        (void) putenv (env);

        env = xmalloc (sizeof "USER=" + strlen (username));
        (void) sprintf (env, "USER=%s", username);
        (void) putenv (env);
    }
#endif /* HAVE_PUTENV */
}
#endif

#ifdef AUTH_SERVER_SUPPORT

extern char *crypt PROTO((const char *, const char *));


/*
 * 0 means no entry found for this user.
 * 1 means entry found and password matches.
 * 2 means entry found, but password does not match.
 *
 * If success, host_user_ptr will be set to point at the system
 * username (i.e., the "real" identity, which may or may not be the
 * CVS username) of this user; caller may free this.  Global
 * CVS_Username will point at an allocated copy of cvs username (i.e.,
 * the username argument below).
 */
static int
check_repository_password (username, password, repository, host_user_ptr)
     char *username, *password, *repository, **host_user_ptr;
{
    int retval = 0;
    FILE *fp;
```

```
    char *filename;
    char *linebuf = NULL;
    size_t linebuf_len;
    int found_it = 0;
    int namelen;

    /* We don't use CVSroot_directory because it hasn't been set yet
     * - our 'repository' argument came from the authentication
     * protocol, not the regular CVS protocol.
     */

    filename = xmalloc (strlen (repository)
                        + 1
                        + strlen (CVSROOTADM)
                        + 1
                        + strlen (CVSROOTADM_PASSWD)
                        + 1);

    (void) sprintf (filename, "%s/%s/%s", repository,
                    CVSROOTADM, CVSROOTADM_PASSWD);

    fp = CVS_FOPEN (filename, "r");
    if (fp == NULL)
    {
        if (!existence_error (errno))
            error (0, errno, "cannot open %s", filename);
        return 0;
    }

    /* Look for a relevant line - one with this user's name. */
    namelen = strlen (username);
    while (getline (&linebuf, &linebuf_len, fp) >= 0)
    {
        if ((strncmp (linebuf, username, namelen) == 0)
            && (linebuf[namelen] == ':'))
        {
            found_it = 1;
            break;
        }
    }
    if (ferror (fp))
        error (0, errno, "cannot read %s", filename);
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", filename);

    /* If found_it != 0, then linebuf contains the information we need. */
    if (found_it)
    {
        char *found_password, *host_user_tmp;

        strtok (linebuf, ":");
        found_password = strtok (NULL, ": \n");
        host_user_tmp = strtok (NULL, ": \n");
        if (host_user_tmp == NULL)
            host_user_tmp = username;

        if (strcmp (found_password, crypt (password, found_password)) == 0)
        {
            /* Give host_user_ptr permanent storage. */
            *host_user_ptr = xstrdup (host_user_tmp);
            retval = 1;
        }
        else
        {
            *host_user_ptr = NULL;
            retval         = 2;
        }
    }
    else
    {
        *host_user_ptr = NULL;
        retval = 0;
    }

    free (filename);
    if (linebuf)
        free (linebuf);

    return retval;
}

/* Return a hosting username if password matches, else NULL. */
static char *
check_password (username, password, repository)
    char *username, *password, *repository;
{
    int rc;
    char *host_user = NULL;
```

```
    /* First we see if this user has a password in the CVS-specific
       password file.  If so, that's enough to authenticate with.  If
       not, we'll check /etc/passwd. */

    rc = check_repository_password (username, password, repository,
                                    &host_user);

    if (rc == 2)
        return NULL;

    /* else */

    if (rc == 1)
    {
        /* host_user already set by reference, so just return. */
        goto handle_return;
    }
    else if (rc == 0 && system_auth)
    {
        /* No cvs password found, so try /etc/passwd. */

        const char *found_passwd = NULL;
        struct passwd *pw;
#ifdef HAVE_GETSPNAM
        struct spwd *spw;

        spw = getspnam (username);
        if (spw != NULL)
        {
            found_passwd = spw->sp_pwdp;
        }
#endif

        if (found_passwd == NULL && (pw = getpwnam (username)) != NULL)
        {
            found_passwd = pw->pw_passwd;
        }

        if (found_passwd == NULL)
        {
            printf ("E Fatal error, aborting.\n\
error 0 %s: no such user\n", username);

            /* I'm doing this manually rather than via error_exit ()
               because I'm not sure whether we want to call server_cleanup.
               Needs more investigation. . . .  */

#ifdef SYSTEM_CLEANUP
            /* Hook for OS-specific behavior, for example socket subsystems on
               NT and OS2 or dealing with windows and arguments on Mac.  */
            SYSTEM_CLEANUP ();
#endif

            exit (EXIT_FAILURE);
        }

        if (*found_passwd)
        {
            /* user exists and has a password */
            host_user = ((! strcmp (found_passwd,
                                    crypt (password, found_passwd)))
                         ? username : NULL);
            goto handle_return;
        }
        else if (password && *password)
        {
            /* user exists and has no system password, but we got
               one as parameter */
            host_user = username;
            goto handle_return;
        }
        else
        {
            /* user exists but has no password at all */
            host_user = NULL;
            goto handle_return;
        }
    }
    else if (rc == 0)
    {
        /* Note that the message _does_ distinguish between the case in
           which we check for a system password and the case in which
           we do not.  It is a real pain to track down why it isn't
           letting you in if it won't say why, and I am not convinced
           that the potential information disclosure to an attacker
           outweighs this.  */
        printf ("error 0 no such user %s in CVSROOT/passwd\n", username);

        /* I'm doing this manually rather than via error_exit ()
           because I'm not sure whether we want to call server_cleanup.
```

```
        Needs more investigation. . . .  */

#ifdef SYSTEM_CLEANUP
        /* Hook for OS-specific behavior, for example socket subsystems on
           NT and OS2 or dealing with windows and arguments on Mac.  */
        SYSTEM_CLEANUP ();
#endif
        exit (EXIT_FAILURE);
    }
    else
    {
        /* Something strange happened.  We don't know what it was, but
           we certainly won't grant authorization. */
        host_user = NULL;
        goto handle_return;
    }

handle_return:
    if (host_user)
    {
        /* Set CVS_Username here, in allocated space.
           It might or might not be the same as host_user. */
        CVS_Username = xmalloc (strlen (username) + 1);
        strcpy (CVS_Username, username);
    }

    return host_user;
}

#endif /* AUTH_SERVER_SUPPORT */

#if defined (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI) || defined (HAVE_KERBEROS)

/* Read username and password from client (i.e., stdin).
   If correct, then switch to run as that user and send an ACK to the
   client via stdout, else send NACK and die. */
void
server_authenticate_connection ()
{
    char *tmp = NULL;
    size_t tmp_allocated = 0;
    char *tmpcopy = NULL;
    char *current = NULL;
    char *next = NULL;
    int verify_and_exit = 0;
    int method;
    int success = 0;
    int result;

    /* The Authentication Protocol.  Client sends:
     *
     *   BEGIN AUTH REQUEST <METHOD>\n
     *   <REPOSITORY>\n
     *   <USERNAME>\n
     *   <PASSWORD>\n
     *   END AUTH REQUEST\n
     *
     * Method is one of pserver/kserver/gserver
     * Server uses above information to authenticate, then sends
     *
     *   I LOVE YOU\n
     *
     * if it grants access, else
     *
     *   I HATE YOU\n
     *
     * if it denies access (and it exits if denying).
     *
     * When the client is "cvs login", the user does not desire actual
     * repository access, but would like to confirm the password with
     * the server.  In this case, the start and stop strings are
     *
     *   BEGIN VERIFICATION REQUEST <METHOD>\n
     *
     *            and
     *
     *   END VERIFICATION REQUEST\n
     *
     * On a verification request, the server's responses are the same
     * (with the obvious semantics), but it exits immediately after
     * sending the response in both cases.
     *
     * Why is the repository sent?  Well, note that the actual
     * client/server protocol can't start up until authentication is
     * successful.  But in order to perform authentication, the server
     * needs to look up the password in the special CVS passwd file,
     * before trying /etc/passwd.  So the client transmits the
     * repository as part of the "authentication protocol".  The
     * repository will be redundantly retransmitted later, but that's no
     * big deal.
```

```
     */

#ifdef SO_KEEPALIVE
    /* Set SO_KEEPALIVE on the socket, so that we don't hang forever
       if the client dies while we are waiting for input.  */
    {
        int on = 1;

        (void) setsockopt (STDIN_FILENO, SOL_SOCKET, SO_KEEPALIVE,
                           (char *) &on, sizeof on);
    }
#endif

    /* Make sure the protocol starts off on the right foot... */
    if (getline (&tmp, &tmp_allocated, stdin) < 0)
        /* FIXME: what?  We could try writing error/eof, but chances
           are the network connection is dead bidirectionally.  log it
           somewhere?  */
        ;

    tmpcopy = xstrdup (tmp);
    error (1, 0, "Out of memory");

    next = tmpcopy;

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
      error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "BEGIN") != 0) {
      error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
      error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "GSSAPI") == 0) {
      method = AUTH_GSSAPI;
    } else if (strcmp (current, "KERBEROS_V4") == 0) {
      method = AUTH_KERBEROS_V4;
    } else if (strcmp (current, "PASSWORD") == 0) {
      method = AUTH_PASSWORD;
    } else {
      error (1, 0, "Bad authentication method: %s", current);
    }

    current = next;
    next = strchr (current, ' ');
    if (next == NULL) {
      error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    *next = '\0';
    next++;

    if (strcmp (current, "AUTHENTICATION") == 0) {
      verify_and_exit = 0;
    } else if (strcmp (current, "VERIFICATION") == 0) {
      verify_and_exit = 0;
    } else {
      error (1, 0, "Bad authentication request: %s", current);
    }

    current = next;
    next = strchr (current, '\n');
    if (next != NULL)
      *next = '\0';

    if (strcmp (current, "REQUEST") != 0) {
      error (1, 0, "Bad authentication protocol start: %s", tmp);
    }

    free (tmpcopy);

    if (method == AUTH_GSSAPI) {
#ifdef HAVE_GSSAPI
      success = gserver_authenticate_connection (verify_and_exit);
#else
      error (1, 0, "GSSAPI authentication not supported by this server");
```

```
#endif
    } else if (method == AUTH_KERBEROS_V4) {
#ifdef HAVE_KERBEROS
        success = kserver_authenticate_connection (verify_and_exit);
#else
        error (1, 0, "Kerberos v4 authentication not supported by this server");
#endif
    } else if (method == AUTH_PASSWORD) {
#ifdef AUTH_SERVER_SUPPORT
        success = pserver_authenticate_connection (verify_and_exit);
#else
        error (1, 0, "Password authentication not supported by this server");
#endif
    }
    else
        error (1, 0, "bad auth protocol start: %s", tmp);

    if (success) {
      /* Make sure the end of the auth request is correct */
      getline (&tmp, &tmp_allocated, stdin);
      if (strcmp (tmp, verify_and_exit ? "END VERIFICATION REQUEST\n" : "END AUTHENTICATION REQUEST\n") != 0) {
        error (1, 0, "Bad authentication protocol end: %s", tmp);
      }

      printf ("I LOVE YOU\n");
      {
        int foo = 0;
        /*      while (foo == 0) {
                }*/
      }
      fflush (stdout);
      result = 0;

    } else {
      printf ("I HATE YOU\n");
      fflush (stdout);
      result = EXIT_FAILURE;
    }

    if (!success || verify_and_exit) {
      /* I'm doing this manually rather than via error_exit ()
         because I'm not sure whether we want to call server_cleanup.
         Needs more investigation....  */

#ifdef SYSTEM_CLEANUP
        /* Hook for OS-specific behavior, for example socket subsystems on
           NT and OS2 or dealing with windows and arguments on Mac.  */
        SYSTEM_CLEANUP ();
#endif

        exit (result);
    }
}

#endif /* defined (AUTH_SERVER_SUPPORT) || defined (HAVE_GSSAPI) || defined (HAVE_KERBEROS) */

#else /* AUTH_SERVER_SUPPORT */

#ifdef AUTH_SERVER_SUPPORT
static int
pserver_authenticate_connection (int verify_and_exit)
{
    char *repository = NULL;
    size_t repository_allocated = 0;
    char *username = NULL;
    size_t username_allocated = 0;
    char *password = NULL;
    size_t password_allocated = 0;
    char *tmp = NULL;
    size_t tmp_allocated = 0;
    int result;

    char *host_user;
    char *descrambled_password;
    /* Get the three important pieces of information in order. */
    /* See above comment about error handling.  */
    getline (&repository, &repository_allocated, stdin);
    getline (&username, &username_allocated, stdin);
    getline (&password, &password_allocated, stdin);

    /* Make them pure. */
    strip_trailing_newlines (repository);
    strip_trailing_newlines (username);
    strip_trailing_newlines (password);

    if (!root_allow_ok (repository)) {
        /* Just give a generic I HATE YOU.  This is because CVS 1.9.10
           and older clients do not support "error".  Once more recent
           clients are more widespread, probably want to fix this (it is
           a real pain to track down why it isn't letting you in if it
```

```
            won't say why, and I am not convinced that the potential
5400        information disclosure to an attacker outweighs this). */
          result = 0;           /* failure */
          goto failure;
        }

        /* OK, now parse the config file, so we can use it to control how
           to check passwords.  If there was an error parsing the config
           file, parse_config already printed an error.  We keep going.
           Why?  Because if we didn't, then there would be no way to check
           in a new CVSROOT/config file to fix the broken one! */
5410    parse_config (repository);

        /* We need the real cleartext before we hash it. */
        descrambled_password = descramble (password);
        host_user = check_password (username, descrambled_password, repository);
        memset (descrambled_password, 0, strlen (descrambled_password));
        free (descrambled_password);
        if (host_user)
        {
            printf ("I LOVE YOU\n");
5420        fflush (stdout);
        }
        else
        {
        i_hate_you:
            printf ("I HATE YOU\n");
            fflush (stdout);
            /* I'm doing this manually rather than via error_exit ()
               because I'm not sure whether we want to call server_cleanup.
               Needs more investigation.... */
5430
#ifdef SYSTEM_CLEANUP
            /* Hook for OS-specific behavior, for example socket subsystems on
               NT and OS2 or dealing with windows and arguments on Mac. */
            SYSTEM_CLEANUP ();
#endif

            exit (EXIT_FAILURE);
        }

5440    /* Don't go any farther if we're just responding to "cvs login". */
        if (verify_and_exit)
        {
#ifdef SYSTEM_CLEANUP
            /* Hook for OS-specific behavior, for example socket subsystems on
               NT and OS2 or dealing with windows and arguments on Mac. */
            SYSTEM_CLEANUP ();
#endif

        if (!host_user) {
5450      result = 0;           /* failure */
          goto failure;
        }

        /* Set Pserver_Repos so that we can check later that the same
           repository is sent in later client/server protocol. */
        Pserver_Repos = xmalloc (strlen (repository) + 1);
        strcpy (Pserver_Repos, repository);

        /* Switch to run as this user. */
5460    if (!verify_and_exit)
        switch_to_user (host_user);
        result = 1;       /* success */

    failure:
        free (tmp);
        free (repository);
        free (username);
        free (password);

5470    return result;
    }
    #endif /* AUTH_SERVER_SUPPORT */


    #ifdef HAVE_KERBEROS
    static int
    kserver_authenticate_connection (int verify_and_exit)
    {
        int status;
5480    char instance[INST_SZ];
        struct sockaddr_in peer;
        struct sockaddr_in laddr;
        int len;
        KTEXT_ST ticket;
        AUTH_DAT auth;
        char version[KRB_SENDAUTH_VLEN];
        char user[ANAME_SZ];
```

```
    strcpy (instance, "*");
    len = sizeof peer;
    if (getpeername (STDIN_FILENO, (struct sockaddr *) &peer, &len) < 0
        || getsockname (STDIN_FILENO, (struct sockaddr *) &laddr,
                        &len) < 0)
    {
        printf ("E Fatal error, aborting.\n\
error %s getpeername or getsockname failed\n", strerror (errno));
        return 0;           /* failure */
    }

#ifdef SO_KEEPALIVE
    /* Set SO_KEEPALIVE on the socket, so that we don't hang forever
       if the client dies while we are waiting for input.  */
    {
        int on = 1;

        (void) setsockopt (STDIN_FILENO, SOL_SOCKET, SO_KEEPALIVE,
                           (char *) &on, sizeof on);
    }
#endif

    status = krb_recvauth (KOPT_DO_MUTUAL, STDIN_FILENO, &ticket, "rcmd",
                           instance, &peer, &laddr, &auth, "", sched,
                           version);
    if (status != KSUCCESS)
    {
        printf ("E Fatal error, aborting.\n\
error 0 kerberos: %s\n", krb_get_err_text(status));
        return 0;           /* failure */
    }

    memcpy (kblock, auth.session, sizeof (C_Block));

    /* Get the local name.  */
    status = krb_kntoln (&auth, user);
    if (status != KSUCCESS)
    {
        printf ("E Fatal error, aborting.\n\
error 0 kerberos: can't get local name: %s\n", krb_get_err_text(status));
        return 0; /* failure */
    }

    /* Switch to run as this user. */
    if (!verify_and_exit)
    switch_to_user (user);
    return 1; /* success */
}
#endif /* HAVE_KERBEROS */

#ifdef HAVE_GSSAPI

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN (256)
#endif

/* Authenticate a GSSAPI connection.  This is called from
   pserver_authenticate_connection, and it handles success and failure
   the same way.  */

static int
gserver_authenticate_connection (int verify_and_exit)
{
    char hostname[MAXHOSTNAMELEN];
    struct hostent *hp;
    gss_buffer_desc tok_in, tok_out;
    char buf[1024];
    OM_uint32 stat_min, ret;
    gss_name_t server_name, client_name;
    gss_cred_id_t server_creds;
    int nbytes;
    gss_OID mechid;

    gethostname (hostname, sizeof hostname);
    hp = gethostbyname (hostname);
    if (hp == NULL)
        error (1, 0, "can't get canonical hostname");

    sprintf (buf, "cvs@%s", hp->h_name);
    tok_in.value = buf;
    tok_in.length = strlen (buf);

    if (gss_import_name (&stat_min, &tok_in, GSS_C_NT_HOSTBASED_SERVICE,
                         &server_name) != GSS_S_COMPLETE)
        error (1, 0, "could not import GSSAPI service name %s", buf);

    /* Acquire the server credential to verify the client's
       authentication.  */
    if (gss_acquire_cred (&stat_min, server_name, 0, GSS_C_NULL_OID_SET,
                          GSS_C_ACCEPT, &server_creds,
```

```
                        NULL, NULL) != GSS_S_COMPLETE)
5580        error (1, 0, "could not acquire GSSAPI server credentials");

    gss_release_name (&stat_min, &server_name);

    /* The client will send us a two byte length followed by that many
       bytes.  */
    if (fread (buf, 1, 2, stdin) != 2)
        error (1, errno, "read of length failed");

    nbytes = ((buf[0] & 0xff) << 8) | (buf[1] & 0xff);
5590    assert (nbytes <= sizeof buf);

    if (fread (buf, 1, nbytes, stdin) != nbytes)
        error (1, errno, "read of data failed");

    gcontext = GSS_C_NO_CONTEXT;
    tok_in.length = nbytes;
    tok_in.value = buf;

    if (gss_accept_sec_context (&stat_min,
5600                            &gcontext,   /* context_handle */
                                server_creds, /* verifier_cred_handle */
                                &tok_in,     /* input_token */
                                NULL,        /* channel bindings */
                                &client_name, /* src_name */
                                &mechid,     /* mech_type */
                                &tok_out,    /* output_token */
                                &ret,
                                NULL,        /* ignore time_rec */
                                NULL)        /* ignore del_cred_handle */
5610        != GSS_S_COMPLETE)
    {
        error (1, 0, "could not verify credentials");
    }

    /* FIXME: Use Kerberos v5 specific code to authenticate to a user.
       We could instead use an authentication to access mapping.  */
    {
        krb5_context kc;
        krb5_principal p;
5620        gss_buffer_desc desc;

        krb5_init_context (&kc);
        if (gss_display_name (&stat_min, client_name, &desc,
                              &mechid) != GSS_S_COMPLETE
            || krb5_parse_name (kc, ((gss_buffer_t) &desc)->value, &p) != 0
            || krb5_aname_to_localname (kc, p, sizeof buf, buf) != 0
            || krb5_kuserok (kc, p, buf) != TRUE)
        {
            error (1, 0, "access denied");
5630        }
        krb5_free_principal (kc, p);
        krb5_free_context (kc);
    }

    if (tok_out.length != 0)
    {
        char cbuf[2];

        cbuf[0] = (tok_out.length >> 8) & 0xff;
5640        cbuf[1] = tok_out.length & 0xff;
        if (fwrite (cbuf, 1, 2, stdout) != 2
            || (fwrite (tok_out.value, 1, tok_out.length, stdout)
                != tok_out.length))
            error (1, errno, "fwrite failed");
    }

        if (!verify_and_exit)
    switch_to_user (buf);

5650        return 1; /* success */
}

#endif /* HAVE_GSSAPI */

#endif /* SERVER_SUPPORT */

#if defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT)

/* This global variable is non-zero if the user requests encryption on
5660   the command line.  */
int cvsencrypt;

/* This global variable is non-zero if the users requests stream
   authentication on the command line.  */
int cvsauthenticate;

#ifdef HAVE_GSSAPI
```

```
/* An buffer interface using GSSAPI.  This is built on top of a
   packetizing buffer.  */

/* This structure is the closure field of the GSSAPI translation
   routines.  */

struct cvs_gssapi_wrap_data
{
    /* The GSSAPI context.  */
    gss_ctx_id_t gcontext;
};

static int cvs_gssapi_wrap_input PROTO((void *, const char *, char *, int));
static int cvs_gssapi_wrap_output PROTO((void *, const char *, char *, int,
                                         int *));

/* Create a GSSAPI wrapping buffer.  We use a packetizing buffer with
   GSSAPI wrapping routines.  */

struct buffer *
cvs_gssapi_wrap_buffer_initialize (buf, input, gcontext, memory)
     struct buffer *buf;
     int input;
     gss_ctx_id_t gcontext;
     void (*memory) PROTO((struct buffer *));
{
    struct cvs_gssapi_wrap_data *gd;

    gd = (struct cvs_gssapi_wrap_data *) xmalloc (sizeof *gd);
    gd->gcontext = gcontext;

    return (packetizing_buffer_initialize
            (buf,
             input ? cvs_gssapi_wrap_input : NULL,
             input ? NULL : cvs_gssapi_wrap_output,
             gd,
             memory));
}

/* Unwrap data using GSSAPI.  */

static int
cvs_gssapi_wrap_input (fnclosure, input, output, size)
     void *fnclosure;
     const char *input;
     char *output;
     int size;
{
    struct cvs_gssapi_wrap_data *gd =
        (struct cvs_gssapi_wrap_data *) fnclosure;
    gss_buffer_desc inbuf, outbuf;
    OM_uint32 stat_min;
    int conf;

    inbuf.value = (void *) input;
    inbuf.length = size;

    if (gss_unwrap (&stat_min, gd->gcontext, &inbuf, &outbuf, &conf, NULL)
        != GSS_S_COMPLETE)
    {
        error (1, 0, "gss_unwrap failed");
    }

    if (outbuf.length > size)
        abort ();

    memcpy (output, outbuf.value, outbuf.length);

    /* The real packet size is stored in the data, so we don't need to
       remember outbuf.length.  */

    gss_release_buffer (&stat_min, &outbuf);

    return 0;
}

/* Wrap data using GSSAPI.  */

static int
cvs_gssapi_wrap_output (fnclosure, input, output, size, translated)
     void *fnclosure;
     const char *input;
     char *output;
     int size;
     int *translated;
{
    struct cvs_gssapi_wrap_data *gd =
        (struct cvs_gssapi_wrap_data *) fnclosure;
    gss_buffer_desc inbuf, outbuf;
    OM_uint32 stat_min;
```

```
    int conf_req, conf;

    inbuf.value = (void *) input;
    inbuf.length = size;

#ifdef ENCRYPTION
    conf_req = cvs_gssapi_encrypt;
#else
    conf_req = 0;
#endif

    if (gss_wrap (&stat_min, gd->gcontext, conf_req, GSS_C_QOP_DEFAULT,
                  &inbuf, &conf, &outbuf) != GSS_S_COMPLETE)
        error (1, 0, "gss_wrap failed");

    /* The packetizing buffer only permits us to add 100 bytes.
       FIXME: I don't know what, if anything, is guaranteed by GSSAPI.
       This may need to be increased for a different GSSAPI
       implementation, or we may need a different algorithm.  */
    if (outbuf.length > size + 100)
        abort ();

    memcpy (output, outbuf.value, outbuf.length);

    *translated = outbuf.length;

    gss_release_buffer (&stat_min, &outbuf);

    return 0;
}

#endif /* HAVE_GSSAPI */

#ifdef ENCRYPTION

#ifdef HAVE_KERBEROS

/* An encryption interface using Kerberos.  This is built on top of a
   packetizing buffer.  */

/* This structure is the closure field of the Kerberos translation
   routines.  */

struct krb_encrypt_data
{
    /* The Kerberos key schedule.  */
    Key_schedule sched;
    /* The Kerberos DES block.  */
    C_Block block;
};

static int krb_encrypt_input PROTO((void *, const char *, char *, int));
static int krb_encrypt_output PROTO((void *, const char *, char *, int,
                                     int *));

/* Create a Kerberos encryption buffer.  We use a packetizing buffer
   with Kerberos encryption translation routines.  */

struct buffer *
krb_encrypt_buffer_initialize (buf, input, sched, block, memory)
     struct buffer *buf;
     int input;
     Key_schedule sched;
     C_Block block;
     void (*memory) PROTO((struct buffer *));
{
    struct krb_encrypt_data *kd;

    kd = (struct krb_encrypt_data *) xmalloc (sizeof *kd);
    memcpy (kd->sched, sched, sizeof (Key_schedule));
    memcpy (kd->block, block, sizeof (C_Block));

    return packetizing_buffer_initialize (buf,
                                          input ? krb_encrypt_input : NULL,
                                          input ? NULL : krb_encrypt_output,
                                          kd,
                                          memory);
}

/* Decrypt Kerberos data.  */

static int
krb_encrypt_input (fnclosure, input, output, size)
     void *fnclosure;
     const char *input;
     char *output;
     int size;
{
    struct krb_encrypt_data *kd = (struct krb_encrypt_data *) fnclosure;
    int tcount;
```

```
5850        des_cbc_encrypt ((C_Block *) input, (C_Block *) output,
                             size, kd->sched, &kd->block, 0);

            /* SIZE is the size of the buffer, which is set by the encryption
               routine.  The packetizing buffer will arrange for the first two
               bytes in the decrypted buffer to be the real (unaligned)
               length.  As a safety check, make sure that the length in the
               buffer corresponds to SIZE.  Note that the length in the buffer
               is just the length of the data.  We must add 2 to account for
               the buffer count itself.  */
5860        tcount = ((output[0] & 0xff) << 8) + (output[1] & 0xff);
            if (((tcount + 2 + 7) & ~7) != size)
              error (1, 0, "Decryption failure");

            return 0;
        }

        /* Encrypt Kerberos data.  */

        static int
5870    krb_encrypt_output (fnclosure, input, output, size, translated)
             void *fnclosure;
             const char *input;
             char *output;
             int size;
             int *translated;
        {
            struct krb_encrypt_data *kd = (struct krb_encrypt_data *) fnclosure;
            int aligned;

5880        /* For security against a known plaintext attack, we should
               initialize any padding bytes to random values.  Instead, we
               just pick up whatever is on the stack, which is at least better
               than using zero.  */

            /* Align SIZE to an 8 byte boundary.  Note that SIZE includes the
               two byte buffer count at the start of INPUT which was added by
               the packetizing buffer.  */
            aligned = (size + 7) & ~7;

5890        /* We use des_cbc_encrypt rather than krb_mk_priv because the
               latter sticks a timestamp in the block, and krb_rd_priv expects
               that timestamp to be within five minutes of the current time.
               Given the way the CVS server buffers up data, that can easily
               fail over a long network connection.  We trust krb_recvauth to
               guard against a replay attack.  */

            des_cbc_encrypt ((C_Block *) input, (C_Block *) output, aligned,
                             kd->sched, &kd->block, 1);

5900        *translated = aligned;

            return 0;
        }

        #endif /* HAVE_KERBEROS */
        #endif /* ENCRYPTION */
        #endif /* defined (CLIENT_SUPPORT) || defined (SERVER_SUPPORT) */

        /* Output LEN bytes at STR.  If LEN is zero, then output up to (not including)
5910       the first '\0' byte.  */

        void
        cvs_output (str, len)
            const char *str;
            size_t len;
        {
            if (len == 0)
                len = strlen (str);
        #ifdef SERVER_SUPPORT
5920        if (error_use_protocol)
            {
                buf_output (saved_output, str, len);
                buf_copy_lines (buf_to_net, saved_output, 'M');
            }
            else if (server_active)
            {
                buf_output (saved_output, str, len);
                buf_copy_lines (protocol, saved_output, 'M');
                buf_send_counted (protocol);
5930        }
            else
        #endif
            {
                size_t written;
                size_t to_write = len;
                const char *p = str;

                /* For symmetry with cvs_outerr we would call fflush (stderr)
```

```
            here.  I guess the assumption is that stderr will be
5940        unbuffered, so we don't need to.  That sounds like a sound
            assumption from the manpage I looked at, but if there was
            something fishy about it, my guess is that calling fflush
            would not produce a significant performance problem.  */

        while (to_write > 0)
        {
            written = fwrite (p, 1, to_write, stdout);
            if (written == 0)
                break;
5950        p += written;
            to_write -= written;
        }
    }
}

/* Output LEN bytes at STR in binary mode.  If LEN is zero, then
   output zero bytes.  */

void
5960 cvs_output_binary (str, len)
    char *str;
    size_t len;
{
#ifdef SERVER_SUPPORT
    if (error_use_protocol || server_active)
    {
        struct buffer *buf;
        char size_text[40];

5970    if (error_use_protocol)
            buf = buf_to_net;
        else
            buf = protocol;

        if (!supported_response ("Mbinary"))
        {
            error (0, 0, "\
this client does not support writing binary files to stdout");
            return;
5980    }

        buf_output0 (buf, "Mbinary\012");
        sprintf (size_text, "%lu\012", (unsigned long) len);
        buf_output0 (buf, size_text);

        /* Not sure what would be involved in using buf_append_data here
           without stepping on the toes of our caller (which is responsible
           for the memory allocation of STR).  */
        buf_output (buf, str, len);
5990
        if (!error_use_protocol)
            buf_send_counted (protocol);
    }
    else
#endif
    {
        size_t written;
        size_t to_write = len;
        const char *p = str;
6000
        /* For symmetry with cvs_outerr we would call fflush (stderr)
           here.  I guess the assumption is that stderr will be
           unbuffered, so we don't need to.  That sounds like a sound
           assumption from the manpage I looked at, but if there was
           something fishy about it, my guess is that calling fflush
           would not produce a significant performance problem.  */
#ifdef USE_SETMODE_STDOUT
        int oldmode;

6010    /* It is possible that this should be the same ifdef as
           USE_SETMODE_BINARY but at least for the moment we keep them
           separate.  Mostly this is just laziness and/or a question
           of what has been tested where.  Also there might be an
           issue of setmode vs. _setmode.  */
        /* The Windows doc says to call setmode only right after startup.
           I assume that what they are talking about can also be helped
           by flushing the stream before changing the mode.  */
        fflush (stdout);
        oldmode = _setmode (_fileno (stdout), OPEN_BINARY);
6020    if (oldmode < 0)
            error (0, errno, "failed to setmode on stdout");
#endif

        while (to_write > 0)
        {
            written = fwrite (p, 1, to_write, stdout);
            if (written == 0)
                break;
```

```
            p += written;
            to_write -= written;
        }
#ifdef USE_SETMODE_STDOUT
        fflush (stdout);
        if (_setmode (_fileno (stdout), oldmode) != OPEN_BINARY)
            error (0, errno, "failed to setmode on stdout");
#endif
    }
}

/* Like CVS_OUTPUT but output is for stderr not stdout.  */

void
cvs_outerr (str, len)
    const char *str;
    size_t len;
{
    if (len == 0)
        len = strlen (str);
#ifdef SERVER_SUPPORT
    if (error_use_protocol)
    {
        buf_output (saved_outerr, str, len);
        buf_copy_lines (buf_to_net, saved_outerr, 'E');
    }
    else if (server_active)
    {
        buf_output (saved_outerr, str, len);
        buf_copy_lines (protocol, saved_outerr, 'E');
        buf_send_counted (protocol);
    }
    else
#endif
    {
        size_t written;
        size_t to_write = len;
        const char *p = str;

        /* Make sure that output appears in order if stdout and stderr
           point to the same place.  For the server case this is taken
           care of by the fact that saved_outerr always holds less
           than a line.  */
        fflush (stdout);

        while (to_write > 0)
        {
            written = fwrite (p, 1, to_write, stderr);
            if (written == 0)
                break;
            p += written;
            to_write -= written;
        }
    }
}

/* Flush stderr.  stderr is normally flushed automatically, of course,
   but this function is used to flush information from the server back
   to the client.  */

void
cvs_flusherr ()
{
#ifdef SERVER_SUPPORT
    if (error_use_protocol)
    {
        /* Flush what we can to the network, but don't block.  */
        buf_flush (buf_to_net, 0);
    }
    else if (server_active)
    {
        /* Send a special count to tell the parent to flush.  */
        buf_send_special_count (protocol, -1);
    }
    else
#endif
        fflush (stderr);
}

/* Make it possible for the user to see what has been written to
   stdout (it is up to the implementation to decide exactly how far it
   should go to ensure this).  */

void
cvs_flushout ()
{
#ifdef SERVER_SUPPORT
    if (error_use_protocol)
    {
        /* Flush what we can to the network, but don't block.  */
```

```
            buf_flush (buf_to_net, 0);
    }
    else if (server_active)
    {
        /* Just do nothing.  This is because the code which
           cvs_flushout replaces, setting stdout to line buffering in
           main.c, didn't get called in the server child process.  But
           in the future it is quite plausible that we'll want to make
           this case work analogously to cvs_flusherr.  */
    }
    else
#endif
        fflush (stdout);
}

/* Output TEXT, tagging it according to TAG.  There are lots more
   details about what TAG means in cvsclient.texi but for the simple
   case (e.g. non-client/server), TAG is just "newline" to output a
   newline (in which case TEXT must be NULL), and any other tag to
   output normal text.

   Note that there is no way to output either \0 or \n as part of TEXT.  */

void
cvs_output_tagged (tag, text)
    char *tag;
    char *text;
{
    if (text != NULL && strchr (text, '\n') != NULL)
        /* Uh oh.  The protocol has no way to cope with this.  For now
           we dump core, although that really isn't such a nice
           response given that this probably can be caused by newlines
           in filenames and other causes other than bugs in CVS.  Note
           that we don't want to turn this into "MT newline" because
           this case is a newline within a tagged item, not a newline
           as extraneous sugar for the user.  */
        assert (0);

    /* Start and end tags don't take any text, per cvsclient.texi.  */
    if (tag[0] == '+' || tag[0] == '-')
        assert (text == NULL);

#ifdef SERVER_SUPPORT
    if (server_active && supported_response ("MT"))
    {
        struct buffer *buf;

        if (error_use_protocol)
            buf = buf_to_net;
        else
            buf = protocol;

        buf_output0 (buf, "MT ");
        buf_output0 (buf, tag);
        if (text != NULL)
        {
            buf_output (buf, " ", 1);
            buf_output0 (buf, text);
        }
        buf_output (buf, "\n", 1);

        if (!error_use_protocol)
            buf_send_counted (protocol);
    }
    else
#endif
    {
        if (strcmp (tag, "newline") == 0)
            cvs_output ("\n", 1);
        else if (text != NULL)
            cvs_output (text, 0);
    }
}
```

A.56 server.h

```
/* Interface between the server and the rest of CVS.  */

/* Miscellaneous stuff which isn't actually particularly server-specific.  */
#ifndef STDIN_FILENO
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
#endif

#ifdef SERVER_SUPPORT

/*
 * Nonzero if we are using the server.  Used by various places to call
 * server-specific functions.
 */
extern int server_active;
extern int server_expanding;

/* Server functions exported to the rest of CVS.  */

/* Run the server.  */
extern int server PROTO((int argc, char **argv));

/* See server.c for description.  */
extern void server_pathname_check PROTO ((char *));

/* We have a new Entries line for a file.  TAG or DATE can be NULL.  */
extern void server_register
    PROTO((char *name, char *version, char *timestamp,
           char *options, char *tag, char *date, char *conflict, char *repository));

/* Set the modification time of the next file sent.  This must be
   followed by a call to server_updated on the same file.  */
extern void server_modtime PROTO ((struct file_info *finfo,
                                   Vers_TS *vers_ts));

/*
 * We want to nuke the Entries line for a file, and (unless
 * server_scratch_entry_only is subsequently called) the file itself.
 */
extern void server_scratch PROTO((char *name));

/*
 * The file which just had server_scratch called on it needs to have only
 * the Entries line removed, not the file itself.
 */
extern void server_scratch_entry_only PROTO((void));

/*
 * We just successfully checked in FILE (which is just the bare
 * filename, with no directory).  REPOSITORY is the directory for the
 * repository.
 */
extern void server_checked_in
    PROTO((char *file, char *update_dir, char *repository));

extern void server_copy_file
    PROTO((char *file, char *update_dir, char *repository, char *newfile));

/* Send the appropriate responses for a file described by FINFO and
   VERS.  This is called after server_register or server_scratch.  In
   the latter case the file is to be removed (and VERS can be NULL).
   In the former case, VERS must be non-NULL, and UPDATED indicates
   whether the file is now up to date (SERVER_UPDATED, yes,
   SERVER_MERGED, no, SERVER_PATCHED, yes, but file is a diff from
   user version to repository version, SERVER_RCS_DIFF, yes, like
   SERVER_PATCHED but with an RCS style diff).  MODE is the mode the
   file should get, or (mode_t) -1 if this should be obtained from the
   file itself.  CHECKSUM is the MD5 checksum of the file, or NULL if
   this need not be sent.  If FILEBUF is not NULL, it holds the
   contents of the file, in which case the file itself may not exist.
   If FILEBUF is not NULL, server_updated will free it.  */
enum server_updated_arg4
{
    SERVER_UPDATED,
    SERVER_MERGED,
    SERVER_PATCHED,
    SERVER_RCS_DIFF
};
#ifdef __STDC__
struct buffer;
#endif

extern void server_updated
    PROTO((struct file_info *finfo, Vers_TS *vers,
           enum server_updated_arg4 updated, mode_t mode,
           unsigned char *checksum, struct buffer *filebuf));
```

```
/* Whether we should send RCS format patches.  */
extern int server_use_rcs_diff PROTO((void));

/* Set the Entries.Static flag.  */
extern void server_set_entstat PROTO((char *update_dir, char *repository));
/* Clear it.  */
extern void server_clear_entstat PROTO((char *update_dir, char *repository));

/* Set or clear a per-directory sticky tag or date.  */
extern void server_set_sticky PROTO((char *update_dir, char *repository,
                                     char *tag, char *date, int nonbranch));
/* Send Template response.  */
extern void server_template PROTO ((char *, char *));

extern void server_update_entries
    PROTO((char *file, char *update_dir, char *repository,
           enum server_updated_arg4 updated));

/* Pointer to a malloc'd string which is the directory which
   the server should prepend to the pathnames which it sends
   to the client.  */
extern char *server_dir;

enum progs {PROG_CHECKIN, PROG_UPDATE};
extern void server_prog PROTO((char *, char *, enum progs));
extern void server_cleanup PROTO((int sig));

#ifdef SERVER_FLOWCONTROL
/* Pause if it's convenient to avoid memory blowout */
extern void server_pause_check PROTO((void));
#endif /* SERVER_FLOWCONTROL */

#ifdef AUTH_SERVER_SUPPORT
extern char *CVS_Username;
extern int system_auth;
#endif /* AUTH_SERVER_SUPPORT */

extern void server_output_not_carried_for_file (struct file_info* finfo, Vers_TS* vers);
extern void server_output_not_carried (char* file, char* rev, char* server, char* root, char* repository);

#endif /* SERVER_SUPPORT */

/* Stuff shared with the client.  */
struct request
{
  /* Name of the request.  */
  char *name;

#ifdef SERVER_SUPPORT
  /*
   * Function to carry out the request.  ARGS is the text of the command
   * after name and, if present, a single space, have been stripped off.
   */
  void (*func) PROTO((char *args));
#endif

  /* Stuff for use by the client.  */
  enum {
      /*
       * Failure to implement this request can imply a fatal
       * error.  This should be set only for commands which were in the
       * original version of the protocol; it should not be set for new
       * commands.
       */
      rq_essential,

      /* Some servers might lack this request.  */
      rq_optional,

      /*
       * Set by the client to one of the following based on what this
       * server actually supports.
       */
      rq_supported,
      rq_not_supported,

      /*
       * If the server supports this request, and we do too, tell the
       * server by making the request.
       */
      rq_enableme
      } status;
};

/* Table of requests ending with an entry with a NULL name.  */
extern struct request requests[];

/* Gzip library, see zlib.c.  */
extern void gunzip_and_write PROTO ((int, char *, unsigned char *, size_t));
extern void read_and_gzip PROTO ((int, char *, unsigned char **, size_t *,
```

```
size_t *, int));
```

A.57 status.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Status Information
 */

#include "cvs.h"

static Dtype status_dirproc PROTO ((void *callerdat, char *dir,
                                    char *repos, char *update_dir,
                                    List *entries));
static int status_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int tag_list_proc PROTO((Node * p, void *closure));

static int local = 0;
static int long_format = 0;
static RCSNode *xrcsnode;

static const char *const status_usage[] =
{
    "Usage: %s %s [-vlR] [files. . .]\n",
    "\t-v\tVerbose format; includes tag information for the file\n",
    "\t-l\tProcess this directory only (not recursive).\n",
    "\t-R\tProcess directories recursively.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
cvsstatus (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int err = 0;

    if (argc == -1)
        usage (status_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+vlR")) != -1)
    {
        switch (c)
        {
            case 'v':
                long_format = 1;
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (status_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    wrap_setup ();

#ifdef CLIENT_SUPPORT
    if (client_active) {
      start_server ();

      ign_setup ();

      if (long_format)
        send_arg("-v");
      if (local)
        send_arg("-l");

      send_file_names (argc, argv, SEND_EXPAND_WILD);

      /* For a while, we tried setting SEND_NO_CONTENTS here so this
         could be a fast operation.  That prevents the
         server from updating our timestamp if the timestamp is
         changed but the file is unmodified.  Worse, it is user-visible
         (shows "locally modified" instead of "up to date" if
         timestamp is changed but file is not).  And there is no good
         workaround (you might not want to run "cvs update"; "cvs -n
```

```
        update" doesn't update CVS/Entries; "cvs diff -brief" or
        something perhaps could be made to work but somehow that
        seems nonintuitive to me even if so).  Given that timestamps
        seem to have the potential to get munged for any number of
        reasons, it seems better to not rely too much on them.  */

      send_files (argc, argv, local, 0, 0);

      send_to_server ("status\012", 0);
      err = get_responses_and_close ();

      return err;
    }
#endif

    /* start the recursion processor */
    err = start_recursion (status_fileproc, (FILESDONEPROC) NULL,
                           status_dirproc, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv, local,
                           W_LOCAL, 0, 1, (char *) NULL, 1);

    return (err);
}

/*
 * display the status of a file
 */
/* ARGSUSED */
static int
status_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    Ctype status;
    char *sstat;
    Vers_TS *vers;

    status = Classify_File (finfo, (char *) NULL, (char *) NULL, (char *) NULL,
                            1, 0, &vers, 0);
    sstat = "Classify Error";
    switch (status)
    {
        case T_UNKNOWN:
            sstat = "Unknown";
            break;
        case T_CHECKOUT:
            sstat = "Needs Checkout";
            break;
#ifdef SERVER_SUPPORT
        case T_PATCH:
            sstat = "Needs Patch";
            break;
#endif
        case T_CONFLICT:
            /* I _think_ that "unresolved" is correct; that if it has
               been resolved then the status will change.  But I'm not
               sure about that.  */
            sstat = "Unresolved Conflict";
            break;
        case T_ADDED:
            sstat = "Locally Added";
            break;
        case T_REMOVED:
            sstat = "Locally Removed";
            break;
        case T_MODIFIED:
            if (vers->ts_conflict)
                sstat = "File had conflicts on merge";
            else
                sstat = "Locally Modified";
            break;
        case T_REMOVE_ENTRY:
            sstat = "Entry Invalid";
            break;
        case T_UPTODATE:
            sstat = "Up-to-date";
            break;
        case T_NEEDS_MERGE:
            sstat = "Needs Merge";
            break;
        case T_TITLE:
            /* I don't think this case can occur here.  Just print
               "Classify Error".  */
            break;
    }

    cvs_output ("\
===================================================================\n", 0);
    if (vers->ts_user == NULL)
    {
```

```
            cvs_output ("File: no file ", 0);
            cvs_output (finfo->file, 0);
            cvs_output ("\t\tStatus: ", 0);
            cvs_output (sstat, 0);
            cvs_output ("\n\n", 0);
        }
        else
        {
            char *buf;
            buf = xmalloc (strlen (finfo->file) + strlen (sstat) + 80);
            sprintf (buf, "File: %-17s\tStatus: %s\n\n", finfo->file, sstat);
            cvs_output (buf, 0);
            free (buf);
        }

        if (vers->vn_user == NULL)
        {
            cvs_output ("   Working revision:\tNo entry for ", 0);
            cvs_output (finfo->file, 0);
            cvs_output ("\n", 0);
        }
        else if (vers->vn_user[0] == '0' && vers->vn_user[1] == '\0')
            cvs_output ("   Working revision:\tNew file!\n", 0);
    #ifdef SERVER_SUPPORT
        else if (server_active)
        {
            cvs_output ("   Working revision:\t", 0);
            cvs_output (vers->vn_user, 0);
            cvs_output ("\n", 0);
        }
    #endif
        else
        {
            cvs_output ("   Working revision:\t", 0);
            cvs_output (vers->vn_user, 0);
            cvs_output ("\t", 0);
            cvs_output (vers->ts_rcs, 0);
            cvs_output ("\n", 0);
        }

        if (vers->vn_rcs == NULL)
            cvs_output ("   Repository revision:\tNo revision control file\n", 0);
        else
        {
            cvs_output ("   Repository revision:\t", 0);
            cvs_output (vers->vn_rcs, 0);
            cvs_output ("\t", 0);
            cvs_output (vers->srcfile->path, 0);
            cvs_output ("\n", 0);
        }

        if (vers->entdata)
        {
            Entnode *edata;

            edata = vers->entdata;
            if (edata->tag)
            {
                if (vers->vn_rcs == NULL)
                {
                    cvs_output ("   Sticky Tag:\t\t", 0);
                    cvs_output (edata->tag, 0);
                    cvs_output (" - MISSING from RCS file!\n", 0);
                }
                else
                {
                    if (isdigit (edata->tag[0]))
                    {
                        cvs_output ("   Sticky Tag:\t\t", 0);
                        cvs_output (edata->tag, 0);
                        cvs_output ("\n", 0);
                    }
                    else
                    {
                        char *branch = NULL;

                        if (RCS_nodeisbranch (finfo->rcs, edata->tag))
                            branch = RCS_whatbranch(finfo->rcs, edata->tag);

                        cvs_output ("   Sticky Tag:\t\t", 0);
                        cvs_output (edata->tag, 0);
                        cvs_output (" (", 0);
                        cvs_output (branch ? "branch" : "revision", 0);
                        cvs_output (": ", 0);
                        cvs_output (branch ? branch : vers->vn_rcs, 0);
                        cvs_output (")\n", 0);

                        if (branch)
                            free (branch);
                    }
```

```
            }
        }
        else if (!really_quiet)
            cvs_output ("   Sticky Tag:\t\t(none)\n", 0);

        if (edata->date)
        {
            cvs_output ("   Sticky Date:\t\t", 0);
            cvs_output (edata->date, 0);
            cvs_output ("\n", 0);
        }
        else if (!really_quiet)
            cvs_output ("   Sticky Date:\t\t(none)\n", 0);

        if (edata->options && edata->options[0])
        {
            cvs_output ("   Sticky Options:\t", 0);
            cvs_output (edata->options, 0);
            cvs_output ("\n", 0);
        }
        else if (!really_quiet)
            cvs_output ("   Sticky Options:\t(none)\n", 0);

        if (long_format && vers->srcfile)
        {
            List *symbols = RCS_symbols(vers->srcfile);

            cvs_output ("\n   Existing Tags:\n", 0);
            if (symbols)
            {
                xrcsnode = finfo->rcs;
                (void) walklist (symbols, tag_list_proc, NULL);
            }
            else
                cvs_output ("\tNo Tags Exist\n", 0);
        }
    }

    cvs_output ("\n", 0);
    freevers_ts (&vers);
    return (0);
}

/*
 * Print a warm fuzzy message
 */
/* ARGSUSED */
static Dtype
status_dirproc (callerdat, dir, repos, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repos;
    char *update_dir;
    List *entries;
{
    if (!quiet)
        error (0, 0, "Examining %s", update_dir);
    return (R_PROCESS);
}

/*
 * Print out a tag and its type
 */
static int
tag_list_proc (p, closure)
    Node *p;
    void *closure;
{
    char *branch = NULL;
    char *buf;

    if (RCS_nodeisbranch (xrcsnode, p->key))
        branch = RCS_whatbranch(xrcsnode, p->key) ;

    buf = xmalloc (80 + strlen (p->key)
                   + (branch ? strlen (branch) : strlen (p->data)));
    sprintf (buf, "\t%-25s\t(%s: %s)\n", p->key,
             branch ? "branch" : "revision",
             branch ? branch : p->data);
    cvs_output (buf, 0);
    free (buf);

    if (branch)
        free (branch);

    return (0);
}
```

A.58 subr.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Various useful functions for the CVS support code.
 */

#include "cvs.h"
#include "getline.h"

extern char *getlogin ();

/*
 * malloc some data and die if it fails
 */
char *
xmalloc (bytes)
    size_t bytes;
{
    char *cp;

    /* Parts of CVS try to xmalloc zero bytes and then free it.  Some
       systems have a malloc which returns NULL for zero byte
       allocations but a free which can't handle NULL, so compensate. */
    if (bytes == 0)
        bytes = 1;

    cp = malloc (bytes);
    if (cp == NULL)
        error (1, 0, "out of memory; can not allocate %lu bytes",
               (unsigned long) bytes);
    return (cp);
}

/*
 * realloc data and die if it fails [I've always wanted to have "realloc" do
 * a "malloc" if the argument is NULL, but you can't depend on it.  Here, I
 * can *force* it.
 */
void *
xrealloc (ptr, bytes)
    void *ptr;
    size_t bytes;
{
    char *cp;

    if (!ptr)
        cp = malloc (bytes);
    else
        cp = realloc (ptr, bytes);

    if (cp == NULL)
        error (1, 0, "can not reallocate %lu bytes", (unsigned long) bytes);
    return (cp);
}

/* Two constants which tune expand_string.  Having MIN_INCR as large
   as 1024 might waste a bit of memory, but it shouldn't be too bad
   (CVS used to allocate arrays of, say, 3000, PATH_MAX (8192, often),
   or other such sizes).  Probably anything which is going to allocate
   memory which is likely to get as big as MAX_INCR shouldn't be doing
   it in one block which must be contiguous, but since getrcskey does
   so, we might as well limit the wasted memory to MAX_INCR or so
   bytes.  */

#define MIN_INCR 1024
#define MAX_INCR (2*1024*1024)

/* *STRPTR is a pointer returned from malloc (or NULL), pointing to *N
   characters of space.  Reallocate it so that points to at least
   NEWSIZE bytes of space.  Gives a fatal error if out of memory;
   if it returns it was successful.  */
void
expand_string (strptr, n, newsize)
    char **strptr;
    size_t *n;
    size_t newsize;
{
    if (*n < newsize)
    {
        while (*n < newsize)
        {
            if (*n < MIN_INCR)
                *n += MIN_INCR;
            else if (*n > MAX_INCR)
```

```
            *n += MAX_INCR;
        else
            *n *= 2;
    }
    *strptr = xrealloc (*strptr, *n);
  }
}

/*
 * Duplicate a string, calling xmalloc to allocate some dynamic space
 */
char *
xstrdup (str)
    const char *str;
{
    char *s;

    if (str == NULL)
        return ((char *) NULL);
    s = xmalloc (strlen (str) + 1);
    (void) strcpy (s, str);
    return (s);
}

/* Remove trailing newlines from STRING, destructively. */
void
strip_trailing_newlines (str)
     char *str;
{
    int len;
    len = strlen (str) - 1;

    while (str[len] == '\n')
        str[len--] = '\0';
}

/* Return the number of levels that path ascends above where it starts.
   For example:
   "../../foo" -> 2
   "foo/../../bar" -> 1
   */
/* FIXME: Should be using ISDIRSEP, last_component, or some other
   mechanism which is more general than just looking at slashes,
   particularly for the client.c caller.  The server.c caller might
   want something different, so be careful.  */
int
pathname_levels (path)
    char *path;
{
    char *p;
    char *q;
    int level;
    int max_level;

    max_level = 0;
    p = path;
    level = 0;
    do
    {
        q = strchr (p, '/');
        if (q != NULL)
            ++q;
        if (p[0] == '.' && p[1] == '.' && (p[2] == '\0' || p[2] == '/'))
        {
            --level;
            if (-level > max_level)
                max_level = -level;
        }
        else if (p[0] == '.' && (p[1] == '\0' || p[1] == '/'))
            ;
        else
            ++level;
        p = q;
    } while (p != NULL);
    return max_level;
}


/* Free a vector, where (*ARGV)[0], (*ARGV)[1], ... (*ARGV)[*PARGC - 1]
   are malloc'd and so is *ARGV itself.  Such a vector is allocated by
   line2argv or expand_wild, for example.  */
void
free_names (pargc, argv)
    int *pargc;
    char **argv;
{
    register int i;

    for (i = 0; i < *pargc; i++)
    {                               /* only do through *pargc */
```

```
        free (argv[i]);
    }
    free (argv);
    *pargc = 0;                         /* and set it to zero when done */
}

/* Convert LINE into arguments separated by SEPCHARS.  Set *ARGC
   to the number of arguments found, and (*ARGV)[0] to the first argument,
   (*ARGV)[1] to the second, etc.  *ARGV is malloc'd and so are each of
   (*ARGV)[0], (*ARGV)[1], ...  Use free_names() to return the memory
   allocated here back to the free pool.  */
void
line2argv (pargc, argv, line, sepchars)
    int *pargc;
    char ***argv;
    char *line;
    char *sepchars;
{
    char *cp;
    /* Could make a case for size_t or some other unsigned type, but
       we'll stick with int to avoid signed/unsigned warnings when
       comparing with *pargc.  */
    int argv_allocated;

    /* Small for testing.  */
    /* argv_allocated must be at least 3 because at some places
       (e.g. checkout_proc) cvs alters argv[2].  */
    argv_allocated = 4;
    *argv = (char **) xmalloc (argv_allocated * sizeof (**argv));

    *pargc = 0;
    for (cp = strtok (line, sepchars); cp; cp = strtok ((char *) NULL, sepchars))
    {
        if (*pargc == argv_allocated)
        {
            argv_allocated *= 2;
            *argv = xrealloc (*argv, argv_allocated * sizeof (**argv));
        }
        (*argv)[*pargc] = xstrdup (cp);
        (*pargc)++;
    }
}

/*
 * Returns the number of dots ('.') found in an RCS revision number
 */
int
numdots (s)
    const char *s;
{
    int dots = 0;

    for (; *s; s++)
    {
        if (*s == '.')
            dots++;
    }
    return (dots);
}

/* Compare revision numbers REV1 and REV2 by consecutive fields.
   Return negative, zero, or positive in the manner of strcmp.  The
   two revision numbers must have the same number of fields, or else
   compare_revnums will return an inaccurate result. */
int
compare_revnums (rev1, rev2)
    const char *rev1;
    const char *rev2;
{
    const char *s, *sp;
    const char *t, *tp;
    char *snext, *tnext;
    int result = 0;

    sp = s = rev1;
    tp = t = rev2;
    while (result == 0)
    {
        result = strtoul (sp, &snext, 10) - strtoul (tp, &tnext, 10);
        if (*snext == '\0' || *tnext == '\0')
            break;
        sp = snext + 1;
        tp = tnext + 1;
    }

    return result;
}

char *
increment_revnum (rev)
```

```
    const char *rev;
{
    char *newrev, *p;
    int lastfield;
    size_t len = strlen (rev);

    newrev = (char *) xmalloc (len + 2);
    memcpy (newrev, rev, len + 1);
    p = strrchr (newrev, '.');
    if (p == NULL)
    {
        free (newrev);
        return NULL;
    }
    lastfield = atoi (++p);
    sprintf (p, "%d", lastfield + 1);

    return newrev;
}

/* Return the username by which the caller should be identified in
   CVS, in contexts such as the author field of RCS files, various
   logs, etc.  */
char *
getcaller ()
{
#ifndef SYSTEM_GETCALLER
    static char *cache;
    struct passwd *pw;
    uid_t uid;
#endif

    /* If there is a CVS username, return it.  */
#ifdef AUTH_SERVER_SUPPORT
    if (CVS_Username != NULL)
        return CVS_Username;
#endif

#ifdef SYSTEM_GETCALLER
    return SYSTEM_GETCALLER ();
#else
    /* Get the caller's login from his uid.  If the real uid is "root"
       try LOGNAME USER or getlogin(). If getlogin() and getpwuid()
       both fail, return the uid as a string.  */

    if (cache != NULL)
        return cache;

    uid = getuid ();
    if (uid == (uid_t) 0)
    {
        char *name;

        /* super-user; try getlogin() to distinguish */
        if (((name = getlogin ()) || (name = getenv("LOGNAME")) ||
             (name = getenv("USER"))) && *name)
        {
            cache = xstrdup (name);
            return cache;
        }
    }
    if ((pw = (struct passwd *) getpwuid (uid)) == NULL)
    {
        char uidname[20];

        (void) sprintf (uidname, "uid%lu", (unsigned long) uid);
        cache = xstrdup (uidname);
        return cache;
    }
    cache = xstrdup (pw->pw_name);
    return cache;
#endif
}

#ifdef lint
#ifndef __GNUC__
/* ARGSUSED */
time_t
get_date (date, now)
    char *date;
    struct timeb *now;
{
    time_t foo = 0;

    return (foo);
}
#endif
#endif

/* Given two revisions, find their greatest common ancestor.  If the
```

```
   two input revisions exist, then rcs guarantees that the gca will
360    exist.  */

char *
gca (rev1, rev2)
    const char *rev1;
    const char *rev2;
{
    int dots;
    char *gca;
    const char *p[2];
370    int j[2];
    char *retval;

    if (rev1 == NULL || rev2 == NULL)
    {
        error (0, 0, "sanity failure in gca");
        abort();
    }

    /* The greatest common ancestor will have no more dots, and numbers
380       of digits for each component no greater than the arguments.  Therefore
       this string will be big enough.  */
    gca = xmalloc (strlen (rev1) + strlen (rev2) + 100);

    /* walk the strings, reading the common parts. */
    gca[0] = '\0';
    p[0] = rev1;
    p[1] = rev2;
    do
    {
390        int i;
        char c[2];
        char *s[2];

        for (i = 0; i < 2; ++i)
        {
            /* swap out the dot */
            s[i] = strchr (p[i], '.');
            if (s[i] != NULL) {
                c[i] = *s[i];
400            }

            /* read an int */
            j[i] = atoi (p[i]);

            /* swap back the dot... */
            if (s[i] != NULL) {
                *s[i] = c[i];
                p[i] = s[i] + 1;
            }
410            else
            {
                /* or mark us at the end */
                p[i] = NULL;
            }

        }

        /* use the lowest. */
        (void) sprintf (gca + strlen (gca), "%d.",
420                        j[0] < j[1] ? j[0] : j[1]);

    } while (j[0] == j[1]
             && p[0] != NULL
             && p[1] != NULL);

    /* back up over that last dot. */
    gca[strlen(gca) - 1] = '\0';

    /* numbers differ, or we ran out of strings.  we're done with the
430       common parts.  */

    dots = numdots (gca);
    if (dots == 0)
    {
        /* revisions differ in trunk major number.  */

        char *q;
        const char *s;

440        s = (j[0] < j[1]) ? p[0] : p[1];

        if (s == NULL)
        {
            /* we only got one number.  this is strange.  */
            error (0, 0, "bad revisions %s or %s", rev1, rev2);
            abort();
        }
        else
```

```
        {
            /* we have a minor number.  use it.  */
            q = gca + strlen (gca);

            *q++ = '.';
            for ( ; *s != '.' && *s != '\0'; )
                *q++ = *s++;

            *q = '\0';
        }
    }
    else if ((dots & 1) == 0)
    {
        /* if we have an even number of dots, then we have a branch.
           remove the last number in order to make it a revision.  */

        char *s;

        s = strrchr(gca, '.');
        *s = '\0';
    }

    retval = xstrdup (gca);
    free (gca);
    return retval;
}

/* Give fatal error if REV is numeric and ARGC,ARGV imply we are
   planning to operate on more than one file.  The current directory
   should be the working directory.  Note that callers assume that we
   will only be checking the first character of REV; it need not have
   '\0' at the end of the tag name and other niceties.  Right now this
   is only called from admin.c, but if people like the concept it probably
   should also be called from diff -r, update -r, get -r, and log -r.  */

void
check_numeric (rev, argc, argv)
    const char *rev;
    int argc;
    char **argv;
{
    if (rev == NULL || !isdigit (*rev))
        return;

    /* Note that the check for whether we are processing more than one
       file is (basically) syntactic; that is, we don't behave differently
       depending on whether a directory happens to contain only a single
       file or whether it contains more than one.  I strongly suspect this
       is the least confusing behavior.  */
    if (argc != 1
        || (!wrap_name_has (argv[0], WRAP_TOCVS) && isdir (argv[0])))
    {
        error (0, 0, "while processing more than one file:");
        error (1, 0, "attempt to specify a numeric revision");
    }
}

/*
 *  Sanity checks and any required fix-up on message passed to RCS via '-m'.
 *  RCS 5.7 requires that a non-total-whitespace, non-null message be provided
 *  with '-m'.  Returns a newly allocated, non-empty buffer with whitespace
 *  stripped from end of lines and end of buffer.
 *
 *  TODO: We no longer use RCS to manage repository files, so maybe this
 *  nonsense about non-empty log fields can be dropped.
 */
char *
make_message_rcslegal (message)
     char *message;
{
    char *dst, *dp, *mp;

    if (message == NULL) message = "";

    /* Strip whitespace from end of lines and end of string. */
    dp = dst = (char *) xmalloc (strlen (message) + 1);
    for (mp = message; *mp != '\0'; ++mp)
    {
        if (*mp == '\n')
        {
            /* At end-of-line; backtrack to last non-space. */
            while (dp > dst && (dp[-1] == ' ' || dp[-1] == '\t'))
                --dp;
        }
        *dp++ = *mp;
    }

    /* Backtrack to last non-space at end of string, and truncate. */
    while (dp > dst && isspace (dp[-1]))
        --dp;
```

```
    *dp = '\0';

    /* After all that, if there was no non-space in the string,
       substitute a non-empty message. */
    if (*dst == '\0')
    {
        free (dst);
        dst = xstrdup ("*** empty log message ***");
    }

    return dst;
}

/* Does the file FINFO contain conflict markers?  The whole concept
   of looking at the contents of the file to figure out whether there are
   unresolved conflicts is kind of bogus (people do want to manage files
   which contain those patterns not as conflict markers), but for now it
   is what we do.  */
int
file_has_markers (finfo)
    const struct file_info *finfo;
{
    FILE *fp;
    char *line = NULL;
    size_t line_allocated = 0;
    int result;

    result = 0;
    fp = CVS_FOPEN (finfo->file, "r");
    if (fp == NULL)
        error (1, errno, "cannot open %s", finfo->fullname);
    while (getline (&line, &line_allocated, fp) > 0)
    {
        if (strncmp (line, RCS_MERGE_PAT, sizeof RCS_MERGE_PAT - 1) == 0)
        {
            result = 1;
            goto out;
        }
    }
    if (ferror (fp))
        error (0, errno, "cannot read %s", finfo->fullname);
out:
    if (fclose (fp) < 0)
        error (0, errno, "cannot close %s", finfo->fullname);
    if (line != NULL)
        free (line);
    return result;
}

/* Read the entire contents of the file NAME into *BUF.
   If NAME is NULL, read from stdin.  *BUF
   is a pointer returned from malloc (or NULL), pointing to *BUFSIZE
   bytes of space.  The actual size is returned in *LEN.  On error,
   give a fatal error.  The name of the file to use in error messages
   (typically will include a directory if we have changed directory)
   is FULLNAME.  MODE is "r" for text or "rb" for binary.  */

void
get_file (name, fullname, mode, buf, bufsize, len)
    const char *name;
    const char *fullname;
    const char *mode;
    char **buf;
    size_t *bufsize;
    size_t *len;
{
    struct stat s;
    size_t nread;
    char *tobuf;
    FILE *e;
    size_t filesize;

    if (name == NULL)
    {
        e = stdin;
        filesize = 100; /* force allocation of minimum buffer */
    }
    else
    {
        if (CVS_LSTAT (name, &s) < 0)
            error (1, errno, "can't stat %s", fullname);

        /* Don't attempt to read special files or symlinks. */
        if (!S_ISREG (s.st_mode))
        {
            *len = 0;
            return;
        }

        /* Convert from signed to unsigned.  */
```

```
        filesize = s.st_size;

        e = open_file (name, mode);
    }

    if (*bufsize < filesize)
    {
        *bufsize = filesize;
        *buf = xrealloc (*buf, *bufsize);
    }

    tobuf = *buf;
    nread = 0;
    while (1)
    {
        size_t got;

        got = fread (tobuf, 1, *bufsize - (tobuf - *buf), e);
        if (ferror (e))
            error (1, errno, "can't read %s", fullname);
        nread += got;
        tobuf += got;

        if (feof (e))
            break;

        /* It's probably paranoid to think S.ST_SIZE might be
           too small to hold the entire file contents, but we
           handle it just in case.  */
        if (tobuf == *buf + *bufsize)
        {
            int c;
            long off;

            c = getc (e);
            if (c == EOF)
                break;
            off = tobuf - *buf;
            expand_string (buf, bufsize, *bufsize + 100);
            tobuf = *buf + off;
            *tobuf++ = c;
            ++nread;
        }
    }

    if (e != stdin && fclose (e) < 0)
        error (0, errno, "cannot close %s", fullname);

    *len = nread;

    /* Force *BUF to be large enough to hold a null terminator. */
    if (*buf != NULL)
    {
        if (nread == *bufsize)
            expand_string (buf, bufsize, *bufsize + 1);
        (*buf)[nread] = '\0';
    }
}
```

A.59 tag.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * Tag
 *
 * Add or delete a symbolic name to an RCS file, or a collection of RCS files.
 * Uses the checked out revision in the current directory.
 */

#include "cvs.h"
#include "savecwd.h"

static int check_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int check_filesdoneproc PROTO ((void *callerdat, int err,
                                       char *repos, char *update_dir,
                                       List *entries));
static int pretag_proc PROTO((char *repository, char *filter));
static void masterlist_delproc PROTO((Node *p));
static void tag_delproc PROTO((Node *p));
static int pretag_list_proc PROTO((Node *p, void *closure));

static Dtype tag_dirproc PROTO ((void *callerdat, char *dir,
                                 char *repos, char *update_dir,
                                 List *entries));
static int tag_fileproc PROTO ((void *callerdat, struct file_info *finfo));
static int tag_filesdoneproc PROTO ((void *callerdat, int err,
                                     char *repos, char *update_dir,
                                     List *entries));

static char *numtag;
static char *date = NULL;
static char *symtag;
static int delete_flag;          /* adding a tag by default */
static int branch_mode;          /* make an automagic "branch" tag */
static int local;                /* recursive by default */
static int force_tag_match = 1;          /* force tag to match by default */
static int force_tag_move;       /* don't force tag to move by default */
static int check_uptodate;       /* no uptodate-check by default */

struct tag_info
{
    Ctype status;
    char *rev;
    char *tag;
    char *options;
};

struct master_lists
{
    List *tlist;
};

static List *mtlist;
static List *tlist;

static const char *const tag_usage[] =
{
    "Usage: %s %s [-lRF] [-b] [-d] [-c] [-r tag|-D date] tag [files. . .]\n",
    "\t-l\tLocal directory only, not recursive.\n",
    "\t-R\tProcess directories recursively.\n",
    "\t-d\tDelete the given tag.\n",
    "\t-r rev\tExisting revision/tag.\n",
    "\t-D\tExisting date.\n",
    "\t-f\tForce a head revision if specified tag not found.\n",
    "\t-b\tMake the tag a \"branch\" tag, allowing concurrent development.\n",
    "\t-F\tMove tag if it already exists.\n",
    "\t-c\tCheck that working files are unmodified.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

int
cvstag (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int err = 0;

    if (argc == -1)
        usage (tag_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+FQqlRcdr:D:bf")) != -1)
```

```
    {
90      switch (c)
        {
            case 'Q':
            case 'q':
#ifdef SERVER_SUPPORT
                /* The CVS 1.5 client sends these options (in addition to
                   Global_option requests), so we must ignore them. */
                if (!server_active)
#endif
                    error (1, 0,
100                        "-q or -Q must be specified before \"%s\"",
                           command_name);
                break;
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case 'd':
110             delete_flag = 1;
                break;
            case 'c':
                check_uptodate = 1;
                break;
            case 'r':
                numtag = optarg;
                break;
            case 'D':
                if (date)
120                 free (date);
                date = Make_Date (optarg);
                break;
            case 'f':
                force_tag_match = 0;
                break;
            case 'b':
                branch_mode = 1;
                break;
            case 'F':
130             force_tag_move = 1;
                break;
            case '?':
            default:
                usage (tag_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;
140

    if (argc == 0)
        usage (tag_usage);
    symtag = argv[0];
    argc--;
    argv++;

    if (date && numtag)
        error (1, 0, "-r and -D options are mutually exclusive");
    if (delete_flag && branch_mode)
150     error (0, 0, "warning: -b ignored with -d options");
    RCS_check_tag (symtag);

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        /* We're the client side.  Fire up the remote server. */
        start_server ();

        ign_setup ();
160

        if (!force_tag_match)
            send_arg ("-f");
        if (local)
            send_arg("-l");
        if (delete_flag)
            send_arg("-d");
        if (check_uptodate)
            send_arg("-c");
        if (branch_mode)
170         send_arg("-b");
        if (force_tag_move)
            send_arg("-F");

        if (numtag)
            option_with_arg ("-r", numtag);
        if (date)
            client_senddate (date);
```

```
        send_arg (symtag);

        send_file_names (argc, argv, SEND_EXPAND_WILD);

        /* SEND_NO_CONTENTS has a mildly bizarre interaction with
           check_uptodate; if the timestamp is modified but the file
           is unmodified, the check will fail, only to have "cvs diff"
           show no differences (and one must do "update" or something to
           reset the client's notion of the timestamp).  */

        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
        send_to_server ("tag\012", 0);
        return get_responses_and_close ();
    }
#endif

    if (numtag != NULL)
        tag_check_valid (numtag, argc, argv, local, 0, "");

    /* check to make sure they are authorized to tag all the
       specified files in the repository */

    mtlist = getlist();
    err = start_recursion (check_fileproc, check_filesdoneproc,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv, local, W_LOCAL, 0, 1,
                           (char *) NULL, 1);

    if (err)
    {
        error (1, 0, "correct the above errors first!");
    }

    /* start the recursion processor */
    err = start_recursion (tag_fileproc, tag_filesdoneproc, tag_dirproc,
                           (DIRLEAVEPROC) NULL, NULL, argc, argv, local,
                           W_LOCAL, 0, 0, (char *) NULL, 1);
    dellist(&mtlist);
    return (err);
}

/* check file that is to be tagged */
/* All we do here is add it to our list */

static int
check_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *xdir;
    Node *p;
    Vers_TS *vers;

    if (check_uptodate)
    {
        Ctype status = Classify_File (finfo, (char *) NULL, (char *) NULL,
                                      (char *) NULL, 1, 0, &vers, 0);
        if ((status != T_UPTODATE) && (status != T_CHECKOUT))
        {
            error (0, 0, "%s is locally modified", finfo->fullname);
            return (1);
        }
    }

    if (finfo->update_dir[0] == '\0')
        xdir = ".";
    else
        xdir = finfo->update_dir;
    if ((p = findnode (mtlist, xdir)) != NULL)
    {
        tlist = ((struct master_lists *) p->data)->tlist;
    }
    else
    {
        struct master_lists *ml;

        tlist = getlist ();
        p = getnode ();
        p->key = xstrdup (xdir);
        p->type = UPDATE;
        ml = (struct master_lists *)
            xmalloc (sizeof (struct master_lists));
        ml->tlist = tlist;
        p->data = (char *) ml;
        p->delproc = masterlist_delproc;
        (void) addnode (mtlist, p);
    }
    /* do tlist */
    p = getnode ();
    p->key = xstrdup (finfo->file);
```

```
    p->type = UPDATE;
    p->delproc = tag_delproc;
    vers = Version_TS (finfo, NULL, NULL, NULL, 0, 0);
    if (vers->srcfile == NULL)
    {
        if (!really_quiet)
            error (0, 0, "nothing known about %s", finfo->file);
        return (1);
    }

    /* Here we duplicate the calculation in tag_fileproc about which
       version we are going to tag.  There probably are some subtle races
       (e.g. numtag is "foo" which gets moved between here and
       tag_fileproc).  */
    if (numtag == NULL && date == NULL)
        p->data = xstrdup (vers->vn_user);
    else
        p->data = RCS_getversion (vers->srcfile, numtag, date,
                                  force_tag_match, NULL);

    if (p->data != NULL)
    {
        int addit = 1;
        char *oversion;

        oversion = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1,
                                   (int *) NULL);
        if (oversion == NULL)
        {
            if (delete_flag)
            {
                addit = 0;
            }
        }
        else if (strcmp(oversion, p->data) == 0)
        {
            addit = 0;
        }
        else if (!force_tag_move)
        {
            addit = 0;
        }
        if (oversion != NULL)
        {
            free(oversion);
        }
        if (!addit)
        {
            free(p->data);
            p->data = NULL;
        }
    }
    freevers_ts(&vers);
    (void) addnode (tlist, p);
    return (0);
}

static int
check_filesdoneproc (callerdat, err, repos, update_dir, entries)
    void *callerdat;
    int err;
    char *repos;
    char *update_dir;
    List *entries;
{
    int n;
    Node *p;

    p = findnode(mtlist, update_dir);
    if (p != NULL)
    {
        tlist = ((struct master_lists *) p->data)->tlist;
    }
    else
    {
        tlist = (List *) NULL;
    }
    if ((tlist == NULL) || (tlist->list->next == tlist->list))
    {
        return (err);
    }
    if ((n = Parse_Info(CVSROOTADM_TAGINFO, repos, pretag_proc, 1)) > 0)
    {
        error (0, 0, "Pre-tag check failed");
        err += n;
    }
    return (err);
}

static int
```

```
pretag_proc(repository, filter)
    char *repository;
    char *filter;
{
    if (filter[0] == '/')
    {
        char *s, *cp;

        s = xstrdup(filter);
        for (cp=s; *cp; cp++)
        {
            if (isspace(*cp))
            {
                *cp = '\0';
                break;
            }
        }
        if (!isfile(s))
        {
            error (0, errno, "cannot find pre-tag filter '%s'", s);
            free(s);
            return (1);
        }
        free(s);
    }
    run_setup (filter);
    run_arg (symtag);
    run_arg (delete_flag ? "del" : force_tag_move ? "mov" : "add");
    run_arg (repository);
    walklist(tlist, pretag_list_proc, NULL);
    return (run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL));
}

static void
masterlist_delproc(p)
    Node *p;
{
    struct master_lists *ml;

    ml = (struct master_lists *)p->data;
    dellist(&ml->tlist);
    free(ml);
    return;
}

static void
tag_delproc(p)
    Node *p;
{
    if (p->data != NULL)
    {
        free(p->data);
        p->data = NULL;
    }
    return;
}

static int
pretag_list_proc(p, closure)
    Node *p;
    void *closure;
{
    if (p->data != NULL)
    {
        run_arg(p->key);
        run_arg(p->data);
    }
    return (0);
}


/*
 * Called to tag a particular file (the currently checked out version is
 * tagged with the specified tag - or the specified tag is deleted).
 */
/* ARGSUSED */
static int
tag_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *version, *oversion;
    char *nversion = NULL;
    char *rev;
    Vers_TS *vers;
    int retcode = 0;

    /* Lock the directory if it is not already locked.  We can't rely
       on tag_dirproc because it won't handle the case where the user
       specifies a list of files on the command line.  */
```

```
    /* We do not need to acquire a full write lock for the tag operation:
       the revisions are obtained from the working directory, so we do not
       require consistency across the entire repository.  However, we do
       need to prevent simultaneous tag operations from interfering with
       each other.  Therefore, we write lock each directory as we enter
       it, and unlock it as we leave it.  */
    lock_dir_for_write (finfo->repository);

    vers = Version_TS (finfo, NULL, NULL, NULL, 0, 0);

    if ((numtag != NULL) || (date != NULL))
    {
        nversion = RCS_getversion(vers->srcfile,
                                  numtag,
                                  date,
                                  force_tag_match,
                                  (int *) NULL);
        if (nversion == NULL)
        {
            freevers_ts (&vers);
            return (0);
        }
    }
    if (delete_flag)
    {

        /*
         * If -d is specified, "force_tag_match" is set, so that this call to
         * RCS_getversion() will return a NULL version string if the symbolic
         * tag does not exist in the RCS file.
         *
         * This is done here because it's MUCH faster than just blindly calling
         * "rcs" to remove the tag... trust me.
         */

        version = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1,
                                  (int *) NULL);
        if (version == NULL || vers->srcfile == NULL)
        {
            freevers_ts (&vers);
            return (0);
        }
        free (version);

        if ((retcode = RCS_deltag(vers->srcfile, symtag)) != 0)
        {
            if (!quiet)
                error (0, retcode == -1 ? errno : 0,
                       "failed to remove tag %s from %s", symtag,
                       vers->srcfile->path);
            freevers_ts (&vers);
            return (1);
        }
        RCS_rewrite (vers->srcfile, NULL, NULL);

        /* warm fuzzies */
        if (!really_quiet)
        {
            cvs_output ("D ", 2);
            cvs_output (finfo->fullname, 0);
            cvs_output ("\n", 1);
        }

        freevers_ts (&vers);
        return (0);
    }

    /*
     * If we are adding a tag, we need to know which version we have checked
     * out and we'll tag that version.
     */
    if (nversion == NULL)
    {
        version = vers->vn_user;
    }
    else
    {
        version = nversion;
    }
    if (version == NULL)
    {
        freevers_ts (&vers);
        return (0);
    }
    else if (strcmp (version, "0") == 0)
    {
        if (!quiet)
            error (0, 0, "couldn't tag added but un-commited file `%s'", finfo->file);
        freevers_ts (&vers);
        return (0);
```

```
    }
540 else if (version[0] == '-')
    {
        if (!quiet)
            error (0, 0, "skipping removed but un-commited file `%s'", finfo->file);
        freevers_ts (&vers);
        return (0);
    }
    else if (vers->srcfile == NULL)
    {
        if (!quiet)
550         error (0, 0, "cannot find revision control file for `%s'", finfo->file);
        freevers_ts (&vers);
        return (0);
    }

    /*
     * As an enhancement for the case where a tag is being re-applied to a
     * large number of files, make one extra call to RCS_getversion to see
     * if the tag is already set in the RCS file.  If so, check to see if it
     * needs to be moved.  If not, do nothing.  This will likely save a lot of
560  * time when simply moving the tag to the "current" head revisions of a
     * module - which I have found to be a typical tagging operation.
     */
    rev = branch_mode ? RCS_magicrev (vers->srcfile, version) : version;
    oversion = RCS_getversion (vers->srcfile, symtag, (char *) NULL, 1,
                               (int *) NULL);
    if (oversion != NULL)
    {
        int isbranch = RCS_nodeisbranch (finfo->rcs, symtag);

570     /*
         * if versions the same and neither old or new are branches don't have
         * to do anything
         */
        if (strcmp (version, oversion) == 0 && !branch_mode && !isbranch)
        {
            free (oversion);
            freevers_ts (&vers);
            return (0);
        }
580
        if (!force_tag_move)
        {
            /* we're NOT going to move the tag */
            cvs_output ("W ", 2);
            cvs_output (finfo->fullname, 0);
            cvs_output (" : ", 0);
            cvs_output (symtag, 0);
            cvs_output (" already exists on ", 0);
            cvs_output (isbranch ? "branch" : "version", 0);
590         cvs_output (" ", 0);
            cvs_output (oversion, 0);
            cvs_output (" : NOT MOVING tag to ", 0);
            cvs_output (branch_mode ? "branch" : "version", 0);
            cvs_output (" ", 0);
            cvs_output (rev, 0);
            cvs_output ("\n", 1);
            free (oversion);
            freevers_ts (&vers);
            return (0);
600     }
        free (oversion);
    }

    {
      int tagged = 0;

      /* Check if this is a remote tag, and if so, tag the file and queue a request
         to tag it on the other side as well */
      if (symtag [0] ==':') {
610     /* format of the remote tag is :host:/cvs/root:tag */
        char* remotetag = symtag;

        remotetag = strchr (remotetag + 1, ':');
        if (remotetag != NULL) {

          remotetag = strchr (remotetag + 1, ':');
          if (remotetag != NULL) {
            char* remote_location;

620         remotetag++;
            /* found the tag */

            remote_location = xstrdup (symtag);
            *strchr (strchr (remote_location + 1, ':') + 1, ':') = '\0';

            retcode = RCS_setremotetag(vers->srcfile, remotetag, rev, remote_location);
            if (retcode == 0) {
              /* If we got this end of the branch, tell the client to go to the other server
```

```
                and create the other end of the branch */
630             char* server = remote_location;
                char* repository = strchr (server, ':');
                char* remote_path;

                *repository = '\0';
                repository ++;

                remote_path = xmalloc (strlen (vers -> srcfile -> path) + strlen (repository) + 1);
                sprintf (remote_path, "%s%s", repository, vers->srcfile->path + strlen (CVSroot_directory));
                *strrchr (remote_path, '/') = '\0';
640
                cvs_output("Create-remote-branch ", 0);
                cvs_output(finfo->fullname, 0);
                cvs_output("\n", 0);
                cvs_output(server, 0);
                cvs_output("\n", 0);
                cvs_output(repository, 0);
                cvs_output("\n", 0);
                cvs_output(remote_path, 0);
                cvs_output("\n", 0);
650             cvs_output(rev, 0);
                cvs_output("\n", 0);
              }
              tagged = 1;
            }
          }
        }

        if (!tagged) {
          retcode = RCS_settag(vers->srcfile, symtag, rev);
660     }
      }

      if (retcode != 0)
      {
          error (1, retcode == -1 ? errno : 0,
                 "failed to set tag %s to revision %s in %s",
                 symtag, rev, vers->srcfile->path);
          freevers_ts (&vers);
          return (1);
670   }
      RCS_rewrite (vers->srcfile, NULL, NULL);

      /* more warm fuzzies */
      if (!really_quiet)
      {
          cvs_output ("T ", 2);
          cvs_output (finfo->fullname, 0);
          cvs_output ("\n", 1);
      }
680
      if (nversion != NULL)
      {
          free (nversion);
      }
      freevers_ts (&vers);
      return (0);
  }

  /* Clear any lock we may hold on the current directory.  */
690
  static int
  tag_filesdoneproc (callerdat, err, repos, update_dir, entries)
      void *callerdat;
      int err;
      char *repos;
      char *update_dir;
      List *entries;
  {
      Lock_Cleanup ();
700
      return (err);
  }

  /*
   * Print a warm fuzzy message
   */
  /* ARGSUSED */
  static Dtype
  tag_dirproc (callerdat, dir, repos, update_dir, entries)
710   void *callerdat;
      char *dir;
      char *repos;
      char *update_dir;
      List *entries;
  {
      if (!quiet)
          error (0, 0, "%s %s", delete_flag ? "Untagging" : "Tagging", update_dir);
      return (R_PROCESS);
```

```
}

/* Code relating to the val-tags file.  Note that this file has no way
   of knowing when a tag has been deleted.  The problem is that there
   is no way of knowing whether a tag still exists somewhere, when we
   delete it some places.  Using per-directory val-tags files (in
   CVSREP) might be better, but that might slow down the process of
   verifying that a tag is correct (maybe not, for the likely cases,
   if carefully done), and/or be harder to implement correctly.  */

struct val_args {
    char *name;
    int found;
};

static int val_fileproc PROTO ((void *callerdat, struct file_info *finfo));

static int
val_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    RCSNode *rcsdata;
    struct val_args *args = (struct val_args *)callerdat;
    char *tag;

    if ((rcsdata = finfo->rcs) == NULL)
        /* Not sure this can happen, after all we passed only
           W_REPOS | W_ATTIC.  */
        return 0;

    tag = RCS_gettag (rcsdata, args->name, 1, (int *) NULL);
    if (tag != NULL)
    {
        /* FIXME: should find out a way to stop the search at this point.  */
        args->found = 1;
        free (tag);
    }
    return 0;
}

static Dtype val_direntproc PROTO ((void *, char *, char *, char *, List *));

static Dtype
val_direntproc (callerdat, dir, repository, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repository;
    char *update_dir;
    List *entries;
{
    /* This is not quite right--it doesn't get right the case of "cvs
       update -d -r foobar" where foobar is a tag which exists only in
       files in a directory which does not exist yet, but which is
       about to be created.  */
    if (isdir (dir))
        return 0;
    return R_SKIP_ALL;
}

/* Check to see whether NAME is a valid tag.  If so, return.  If not
   print an error message and exit.  ARGC, ARGV, LOCAL, and AFLAG specify
   which files we will be operating on.

   REPOSITORY is the repository if we need to cd into it, or NULL if
   we are already there, or "" if we should do a W_LOCAL recursion.
   Sorry for three cases, but the "" case is needed in case the
   working directories come from diverse parts of the repository, the
   NULL case avoids an unneccesary chdir, and the non-NULL, non-""
   case is needed for checkout, where we don't want to chdir if the
   tag is found in CVSROOTADM_VALTAGS, but there is not (yet) any
   local directory.  */
void
tag_check_valid (name, argc, argv, local, aflag, repository)
    char *name;
    int argc;
    char **argv;
    int local;
    int aflag;
    char *repository;
{
    DBM *db;
    char *valtags_filename;
    int err;
    datum mytag;
    struct val_args the_val_args;
    struct saved_cwd cwd;
    int which;

    /* Numeric tags require only a syntactic check.  */
```

```
    if (isdigit (name[0]))
    {
        char *p;
        for (p = name; *p != '\0'; ++p)
        {
            if (!(isdigit (*p) || *p == '.'))
                error (1, 0, "\
Numeric tag %s contains characters other than digits and '.'", name);
        }
        return;
    }

    /* Special tags are always valid.  */
    if (strcmp (name, TAG_BASE) == 0
        || strcmp (name, TAG_HEAD) == 0)
        return;

    /* FIXME: This routine doesn't seem to do any locking whatsoever
       (and it is called from places which don't have locks in place).
       If two processes try to write val-tags at the same time, it would
       seem like we are in trouble.  */

    mytag.dptr = name;
    mytag.dsize = strlen (name);

    valtags_filename = xmalloc (strlen (CVSroot_directory)
                                + sizeof CVSROOTADM
                                + sizeof CVSROOTADM_VALTAGS + 20);
    strcpy (valtags_filename, CVSroot_directory);
    strcat (valtags_filename, "/");
    strcat (valtags_filename, CVSROOTADM);
    strcat (valtags_filename, "/");
    strcat (valtags_filename, CVSROOTADM_VALTAGS);
    db = dbm_open (valtags_filename, O_RDWR, 0666);
    if (db == NULL)
    {
        if (!existence_error (errno))
            error (1, errno, "cannot read %s", valtags_filename);

        /* If the file merely fails to exist, we just keep going and create
           it later if need be.  */
    }
    else
    {
        datum val;

        val = dbm_fetch (db, mytag);
        if (val.dptr != NULL)
        {
            /* Found.  The tag is valid.  */
            dbm_close (db);
            free (valtags_filename);
            return;
        }
        /* FIXME: should check errors somehow (add dbm_error to myndbm.c?).  */
    }

    /* We didn't find the tag in val-tags, so look through all the RCS files
       to see whether it exists there.  Yes, this is expensive, but there
       is no other way to cope with a tag which might have been created
       by an old version of CVS, from before val-tags was invented.

       Since we need this code anyway, we also use it to create
       entries in val-tags in general (that is, the val-tags entry
       will get created the first time the tag is used, not when the
       tag is created).  */

    the_val_args.name = name;
    the_val_args.found = 0;

    which = W_REPOS | W_ATTIC;

    if (repository != NULL)
    {
        if (repository[0] == '\0')
            which |= W_LOCAL;
        else
        {
            if (save_cwd (&cwd))
                error_exit ();
            if ( CVS_CHDIR (repository) < 0)
                error (1, errno, "cannot change to %s directory", repository);
        }
    }

    err = start_recursion (val_fileproc, (FILESDONEPROC) NULL,
                           val_direntproc, (DIRLEAVEPROC) NULL,
                           (void *)&the_val_args,
                           argc, argv, local, which, aflag,
                           1, NULL, 1);
```

```
    if (repository != NULL && repository[0] != '\0')
    {
        if (restore_cwd (&cwd, NULL))
            exit (EXIT_FAILURE);
        free_cwd (&cwd);
    }

    if (!the_val_args.found)
        error (1, 0, "no such tag %s", name);
    else
    {
        /* The tags is valid but not mentioned in val-tags.  Add it.  */
        datum value;

        if (noexec)
        {
            if (db != NULL)
                dbm_close (db);
            free (valtags_filename);
            return;
        }

        if (db == NULL)
        {
            mode_t omask;
            omask = umask (cvsumask);
            db = dbm_open (valtags_filename, O_RDWR | O_CREAT | O_TRUNC, 0666);
            (void) umask (omask);

            if (db == NULL)
            {
                error (0, errno, "cannot create %s", valtags_filename);
                free (valtags_filename);
                return;
            }
        }
        value.dptr = "y";
        value.dsize = 1;
        if (dbm_store (db, mytag, value, DBM_REPLACE) < 0)
            error (0, errno, "cannot store %s into %s", name,
                   valtags_filename);
        dbm_close (db);
    }
    free (valtags_filename);
}

/*
 * Check whether a join tag is valid.  This is just like
 * tag_check_valid, but we must stop before the colon if there is one.
 */

void
tag_check_valid_join (join_tag, argc, argv, local, aflag, repository)
    char *join_tag;
    int argc;
    char **argv;
    int local;
    int aflag;
    char *repository;
{
    char *c, *s;

    c = xstrdup (join_tag);
    s = strchr (c, ':');
    if (s != NULL)
    {
        if (isdigit (join_tag[0]))
            error (1, 0,
                   "Numeric join tag %s may not contain a date specifier",
                   join_tag);

        *s = '\0';
    }

    tag_check_valid (c, argc, argv, local, aflag, repository);

    free (c);
}
```

A.60 update.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 *
 * "update" updates the version in the present directory with respect to the RCS
 * repository.  The present version must have been created by "checkout".  The
 * user can keep up-to-date by calling "update" whenever he feels like it.
 *
 * The present version can be committed by "commit", but this keeps the version
 * in tact.
 *
 * Arguments following the options are taken to be file names to be updated,
 * rather than updating the entire directory.
 *
 * Modified or non-existent RCS files are checked out and reported as U
 * <user_file>
 *
 * Modified user files are reported as M <user_file>.  If both the RCS file and
 * the user file have been modified, the user file is replaced by the result
 * of rcsmerge, and a backup file is written for the user in .#file.version.
 * If this throws up irreconcilable differences, the file is reported as C
 * <user_file>, and as M <user_file> otherwise.
 *
 * Files added but not yet committed are reported as A <user_file>.  Files
 * removed but not yet committed are reported as R <user_file>.
 *
 * If the current directory contains subdirectories that hold concurrent
 * versions, these are updated too.  If the -d option was specified, new
 * directories added to the repository are automatically created and updated
 * as well.
 */

#include "cvs.h"
#include "savecwd.h"
#ifdef SERVER_SUPPORT
#include "md5.h"
#endif
#include "watch.h"
#include "fileattr.h"
#include "edit.h"
#include "getline.h"
#include "buffer.h"
#include "hardlink.h"

static int checkout_file PROTO ((struct file_info *finfo, Vers_TS *vers_ts,
                                 int adding, int merging, int update_server));
#ifdef SERVER_SUPPORT
static void checkout_to_buffer PROTO ((void *, const char *, size_t));
#endif
#ifdef SERVER_SUPPORT
static int patch_file PROTO ((struct file_info *finfo,
                              Vers_TS *vers_ts,
                              int *docheckout, struct stat *file_info,
                              unsigned char *checksum));
static void patch_file_write PROTO ((void *, const char *, size_t));
#endif
static int merge_file PROTO ((struct file_info *finfo, Vers_TS *vers));
static int scratch_file PROTO((struct file_info *finfo));
static Dtype update_dirent_proc PROTO ((void *callerdat, char *dir,
                                        char *repository, char *update_dir,
                                        List *entries));
static int update_dirleave_proc PROTO ((void *callerdat, char *dir,
                                        int err, char *update_dir,
                                        List *entries));
static int update_fileproc PROTO ((void *callerdat, struct file_info *));
static int update_filesdone_proc PROTO ((void *callerdat, int err,
                                         char *repository, char *update_dir,
                                         List *entries));
#ifdef PRESERVE_PERMISSIONS_SUPPORT
static int get_linkinfo_proc PROTO ((void *callerdat, struct file_info *));
#endif
static void write_letter PROTO ((struct file_info *finfo, int letter));
static void join_file PROTO ((struct file_info *finfo, Vers_TS *vers_ts));

static char *options = NULL;
static char *tag = NULL;
static char *date = NULL;
/* This is a bit of a kludge.  We call WriteTag at the beginning
   before we know whether nonbranch is set or not.  And then at the
   end, once we have the right value for nonbranch, we call WriteTag
   again.  I don't know whether the first call is necessary or not.
   rewrite_tag is nonzero if we are going to have to make that second
   call.  */
static int rewrite_tag;
static int nonbranch;
```

```
90  /* If we set the tag or date for a subdirectory, we use this to undo
       the setting.  See update_dirent_proc.  */
    static char *tag_update_dir;

    static char *join_rev1, *date_rev1;
    static char *join_rev2, *date_rev2;
    static int aflag = 0;
    static int force_tag_match = 1;
    static int update_build_dirs = 0;
    static int update_prune_dirs = 0;
100 static int pipeout = 0;
    #ifdef SERVER_SUPPORT
    static int patches = 0;
    static int rcs_diff_patches = 0;
    #endif
    static List *ignlist = (List *) NULL;
    static time_t last_register_time;
    static const char *const update_usage[] =
    {
        "Usage: %s %s [-APdflRp] [-k kopt] [-r rev|-D date] [-j rev]\n",
110     "    [-I ign] [-W spec] [files...]\n",
        "\t-A\tReset any sticky tags/date/kopts.\n",
        "\t-P\tPrune empty directories.\n",
        "\t-d\tBuild directories, like checkout does.\n",
        "\t-f\tForce a head revision match if tag/date not found.\n",
        "\t-l\tLocal directory only, no recursion.\n",
        "\t-R\tProcess directories recursively.\n",
        "\t-p\tSend updates to standard output (avoids stickiness).\n",
        "\t-k kopt\tUse RCS kopt -k option on checkout.\n",
        "\t-r rev\tUpdate using specified revision/tag (is sticky).\n",
120     "\t-D date\tSet date to update from (is sticky).\n",
        "\t-j rev\tMerge in changes made between current revision and rev.\n",
        "\t-I ign\tMore files to ignore (! to reset).\n",
        "\t-W spec\tWrappers specification line.\n",
        "(Specify the --help global option for a list of other help options)\n",
        NULL
    };

    /*
     * update is the argv,argc based front end for arg parsing
130  */
    int
    update (argc, argv)
        int argc;
        char **argv;
    {
        int c, err;
        int local = 0;                  /* recursive by default */
        int which;                      /* where to look for files and dirs */

140     if (argc == -1)
            usage (update_usage);

        ign_setup ();
        wrap_setup ();

        /* parse the args */
        optind = 0;
        while ((c = getopt (argc, argv, "+ApPflRQqduk:r:D:j:I:W:")) != -1)
        {
150         switch (c)
            {
                case 'A':
                    aflag = 1;
                    break;
                case 'I':
                    ign_add (optarg, 0);
                    break;
                case 'W':
                    wrap_add (optarg, 0);
160                 break;
                case 'k':
                    if (options)
                        free (options);
                    options = RCS_check_kflag (optarg);
                    break;
                case 'l':
                    local = 1;
                    break;
                case 'R':
170                 local = 0;
                    break;
                case 'Q':
                case 'q':
    #ifdef SERVER_SUPPORT
                    /* The CVS 1.5 client sends these options (in addition to
                       Global_option requests), so we must ignore them.  */
                    if (!server_active)
    #endif
```

```
                error (1, 0,
                       "-q or -Q must be specified before \"%s\"",
                       command_name);
            break;
        case 'd':
            update_build_dirs = 1;
            break;
        case 'f':
            force_tag_match = 0;
            break;
        case 'r':
            tag = optarg;
            break;
        case 'D':
            date = Make_Date (optarg);
            break;
        case 'P':
            update_prune_dirs = 1;
            break;
        case 'p':
            pipeout = 1;
            noexec = 1;          /* so no locks will be created */
            break;
        case 'j':
            if (join_rev2)
                error (1, 0, "only two -j options can be specified");
            if (join_rev1)
                join_rev2 = optarg;
            else
                join_rev1 = optarg;
            break;
        case 'u':
#ifdef SERVER_SUPPORT
            if (server_active)
            {
                patches = 1;
                rcs_diff_patches = server_use_rcs_diff ();
            }
            else
#endif
                usage (update_usage);
            break;
        case '?':
        default:
            usage (update_usage);
            break;
        }
    }
    /* Don't mess with this if we are fetching a remote */
    if (client_active && !handling_remotes)
      first_file_arg = argc;
    argc -= optind;
    argv += optind;

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        int pass;

        /* The first pass does the regular update.  If we receive at least
           one patch which failed, we do a second pass and just fetch
           those files whose patches failed.  */
        pass = 1;
        do
        {
            int status;

            start_server ();

            if (local)
                send_arg("-l");
            if (update_build_dirs)
                send_arg("-d");
            if (pipeout)
                send_arg("-p");
            if (!force_tag_match)
                send_arg("-f");
            if (aflag)
                send_arg("-A");
            if (update_prune_dirs)
                send_arg("-P");
            client_prune_dirs = update_prune_dirs;
            option_with_arg ("-r", tag);
            if (options && options[0] != '\0')
                send_arg (options);
            if (date)
                client_senddate (date);
            if (join_rev1)
                option_with_arg ("-j", join_rev1);
            if (join_rev2)
```

```
            option_with_arg ("-j", join_rev2);
        wrap_send ();

        /* If the server supports the command "update-patches", that means
           that it knows how to handle the -u argument to update, which
           means to send patches instead of complete files.

           We don't send -u if failed_patches != NULL, so that the
           server doesn't try to send patches which will just fail
           again.  At least currently, the client also clobbers the
           file and tells the server it is lost, which also will get
           a full file instead of a patch, but it seems clean to omit
           -u.  */
        if (failed_patches == NULL)
        {
            if (supported_request ("update-patches"))
                send_arg ("-u");
        }

        if (failed_patches == NULL)
        {
            send_file_names (argc, argv, SEND_EXPAND_WILD);
            /* If noexec, probably could be setting SEND_NO_CONTENTS.
               Same caveats as for "cvs status" apply.  */
            send_files (argc, argv, local, aflag,
                        update_build_dirs ? SEND_BUILD_DIRS : 0);
        }
        else
        {
            int i;

            (void) printf ("%s client: refetching unpatchable files\n",
                           program_name);

            if (toplevel_wd != NULL
                && CVS_CHDIR (toplevel_wd) < 0)
            {
                error (1, errno, "could not chdir to %s", toplevel_wd);
            }

            for (i = 0; i < failed_patches_count; i++)
                (void) unlink_file (failed_patches[i]);
            send_file_names (failed_patches_count, failed_patches, 0);
            send_files (failed_patches_count, failed_patches, local,
                        aflag, update_build_dirs ? SEND_BUILD_DIRS : 0);
        }

        failed_patches = NULL;
        failed_patches_count = 0;

        send_to_server ("update\012", 0);

        status = get_responses_and_close ();

        /* If there are any conflicts, the server will return a
           non-zero exit status.  If any patches failed, we still
           want to run the update again.  We use a pass count to
           avoid an endless loop.  */

        /* Notes: (1) assuming that status != 0 implies a
           potential conflict is the best we can cleanly do given
           the current protocol.  I suppose that trying to
           re-fetch in cases where there was a more serious error
           is probably more or less harmless, but it isn't really
           ideal.  (2) it would be nice to have a testsuite case for the
           conflict-and-patch-failed case.  */

        if (status != 0
            && (failed_patches == NULL || pass > 1))
        {
            return status;
        }

        ++pass;
      } while (failed_patches != NULL);

      return 0;
    }
#endif

    if (tag != NULL)
        tag_check_valid (tag, argc, argv, local, aflag, "");
    if (join_rev1 != NULL)
        tag_check_valid_join (join_rev1, argc, argv, local, aflag, "");
    if (join_rev2 != NULL)
        tag_check_valid_join (join_rev2, argc, argv, local, aflag, "");

    /*
     * If we are updating the entire directory (for real) and building dirs
     * as we go, we make sure there is no static entries file and write the
```

```
     * tag file as appropriate
360  */
    if (argc <= 0 && !pipeout)
    {
        if (update_build_dirs)
        {
            if (unlink_file (CVSADM_ENTSTAT) < 0 && ! existence_error (errno))
                error (1, errno, "cannot remove file %s", CVSADM_ENTSTAT);
#ifdef SERVER_SUPPORT
            if (server_active)
                server_clear_entstat (".", Name_Repository (NULL, NULL));
370 #endif
        }

        /* keep the CVS/Tag file current with the specified arguments */
        if (aflag || tag || date)
        {
            WriteTag ((char *) NULL, tag, date, 0,
                      ".", Name_Repository (NULL, NULL));
            rewrite_tag = 1;
            nonbranch = 0;
380     }
    }

    /* look for files/dirs locally and in the repository */
    which = W_LOCAL | W_REPOS;

    /* look in the attic too if a tag or date is specified */
    if (tag != NULL || date != NULL || joining())
        which |= W_ATTIC;

390 /* call the command line interface */
    err = do_update (argc, argv, options, tag, date, force_tag_match,
                     local, update_build_dirs, aflag, update_prune_dirs,
                     pipeout, which, join_rev1, join_rev2, (char *) NULL);

    /* free the space Make_Date allocated if necessary */
    if (date != NULL)
        free (date);

    if (options != NULL) {
400   free (options);
      options = NULL;
    }

    return (err);
}

/*
 * Command line interface to update (used by checkout)
 */
410 int
do_update (argc, argv, xoptions, xtag, xdate, xforce, local, xbuild, xaflag,
           xprune, xpipeout, which, xjoin_rev1, xjoin_rev2, preload_update_dir)
    int argc;
    char **argv;
    char *xoptions;
    char *xtag;
    char *xdate;
    int xforce;
    int local;
420 int xbuild;
    int xaflag;
    int xprune;
    int xpipeout;
    int which;
    char *xjoin_rev1;
    char *xjoin_rev2;
    char *preload_update_dir;
{
    int err = 0;
430 char *cp;

    /* fill in the statics */
    options = xoptions;
    tag = xtag;
    date = xdate;
    force_tag_match = xforce;
    update_build_dirs = xbuild;
    aflag = xaflag;
    update_prune_dirs = xprune;
440 pipeout = xpipeout;

    /* setup the join support */
    join_rev1 = xjoin_rev1;
    join_rev2 = xjoin_rev2;
    if (join_rev1 && (cp = strchr (join_rev1, ':')) != NULL)
    {
        *cp++ = '\0';
        date_rev1 = Make_Date (cp);
```

```
    }
    else
        date_rev1 = (char *) NULL;
    if (join_rev2 && (cp = strchr (join_rev2, ':')) != NULL)
    {
        *cp++ = '\0';
        date_rev2 = Make_Date (cp);
    }
    else
        date_rev2 = (char *) NULL;

#ifdef PRESERVE_PERMISSIONS_SUPPORT
    if (preserve_perms)
    {
        /* We need to do an extra recursion, bleah.  It's to make sure
           that we know as much as possible about file linkage. */
        hardlist = getlist();
        working_dir = xgetwd();  /* save top-level working dir */

        /* FIXME-twp: the arguments to start_recursion make me dizzy.  This
           function call was copied from the update_fileproc call that
           follows it; someone should make sure that I did it right. */
        err = start_recursion (get_linkinfo_proc, (FILESDONEPROC) NULL,
                               (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                               argc, argv, local, which, aflag, 1,
                               preload_update_dir, 1);
        if (err)
            return (err);

        /* FIXME-twp: at this point we should walk the hardlist
           and update the 'links' field of each hardlink_info struct
           to list the files that are linked on dist.  That would make
           it easier & more efficient to compare the disk linkage with
           the repository linkage (a simple strcmp). */
    }
#endif

    /* call the recursion processor */
    err = start_recursion (update_fileproc, update_filesdone_proc,
                           update_dirent_proc, update_dirleave_proc, NULL,
                           argc, argv, local, which, aflag, 1,
                           preload_update_dir, 1);

    /* see if we need to sleep before returning */
    if (last_register_time)
    {
        time_t now;

        (void) time (&now);
        if (now == last_register_time)
            sleep (1);                  /* to avoid time-stamp races */
    }

    return (err);
}

#ifdef PRESERVE_PERMISSIONS_SUPPORT
/*
 * The get_linkinfo_proc callback adds each file to the hardlist
 * (see hardlink.c).
 */

static int
get_linkinfo_proc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *fullpath;
    Node *linkp;
    struct hardlink_info *hlinfo;

    /* Get the full pathname of the current file. */
    fullpath = xmalloc (strlen(working_dir) +
                        strlen(finfo->fullname) + 2);
    sprintf (fullpath, "%s/%s", working_dir, finfo->fullname);

    /* To permit recursing into subdirectories, files
       are keyed on the full pathname and not on the basename. */
    linkp = lookup_file_by_inode (fullpath);
    if (linkp == NULL)
    {
        /* The file isn't on disk; we are probably restoring
           a file that was removed. */
        return 0;
    }

    /* Create a new, empty hardlink_info node. */
    hlinfo = (struct hardlink_info *)
        xmalloc (sizeof (struct hardlink_info));
```

```
    hlinfo->status = (Ctype) 0; /* is this dumb? */
    hlinfo->checked_out = 0;

    linkp->data = (char *) hlinfo;

    return 0;
}
#endif

/*
 * This is the callback proc for update.  It is called for each file in each
 * directory by the recursion code.  The current directory is the local
 * instantiation.  file is the file name we are to operate on. update_dir is
 * set to the path relative to where we started (for pretty printing).
 * repository is the repository. entries and srcfiles are the pre-parsed
 * entries and source control files.
 *
 * This routine decides what needs to be done for each file and does the
 * appropriate magic for checkout
 */
static int
update_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    int retval;
    Ctype status;
    Vers_TS *vers;
    int resurrecting;

    resurrecting = 0;

    status = Classify_File (finfo, tag, date, options, force_tag_match,
                            aflag, &vers, pipeout);

    /* Keep track of whether TAG is a branch tag.
       Note that if it is a branch tag in some files and a nonbranch tag
       in others, treat it as a nonbranch tag.  It is possible that case
       should elicit a warning or an error.  */
    if (rewrite_tag
        && tag != NULL
        && finfo->rcs != NULL)
    {
        char *rev = RCS_getversion (finfo->rcs, tag, NULL, 1, NULL);
        if (rev != NULL
            && !RCS_nodeisbranch (finfo->rcs, tag))
            nonbranch = 1;
        if (rev != NULL)
            free (rev);
    }

    if (pipeout)
    {
        /*
         * We just return success without doing anything if any of the really
         * funky cases occur
         *
         * If there is still a valid RCS file, do a regular checkout type
         * operation
         */
        switch (status)
        {
            case T_UNKNOWN:  /* unknown file was explicitly asked
                              * about */
            case T_REMOVE_ENTRY: /* needs to be un-registered */
            case T_ADDED:        /* added but not committed */
                retval = 0;
                break;
            case T_CONFLICT:  /* old punt-type errors */
                retval = 1;
                break;
            case T_UPTODATE: /* file was already up-to-date */
            case T_NEEDS_MERGE: /* needs merging */
            case T_MODIFIED:  /* locally modified */
            case T_REMOVED:   /* removed but not committed */
            case T_CHECKOUT: /* needs checkout */
#ifdef SERVER_SUPPORT
            case T_PATCH:        /* needs patch */
#endif
                retval = checkout_file (finfo, vers, 0, 0, 0);
                break;

            default:                 /* can't ever happen :-) */
                error (0, 0,
                       "unknown file status %d for file %s", status, finfo->file);
                retval = 0;
                break;
        }
    }
    else
```

```
    {
630     switch (status)
        {
            case T_UNKNOWN:  /* unknown file was explicitly asked
                              * about */
            case T_UPTODATE: /* file was already up-to-date */
                retval = 0;
                break;
            case T_CONFLICT:  /* old punt-type errors */
                retval = 1;
                write_letter (finfo, 'C');
640             break;
            case T_NEEDS_MERGE: /* needs merging */
                if (noexec)
                {
                    retval = 1;
                    write_letter (finfo, 'C');
                }
                else
                {
                    retval = merge_file (finfo, vers);
650             }
                break;
            case T_MODIFIED:  /* locally modified */
                retval = 0;
                if (vers->ts_conflict)
                {
                    char *filestamp;
                    int retcode;

                    /*
660                  * If the timestamp has changed and no conflict indicators
                     * are found, it isn't a 'C' any more.
                     */
#ifdef SERVER_SUPPORT
                    if (server_active)
                        retcode = vers->ts_conflict[0] != '=';
                    else {
                        filestamp = time_stamp (finfo->file);
                        retcode = strcmp (vers->ts_conflict, filestamp);
                        free (filestamp);
670                 }
#else
                    filestamp = time_stamp (finfo->file);
                    retcode = strcmp (vers->ts_conflict, filestamp);
                    free (filestamp);
#endif

                    if (retcode)
                    {
                        /* The timestamps differ.  But if there are conflict
680                        markers print 'C' anyway.  */
                        retcode = !file_has_markers (finfo);
                    }

                    if (!retcode)
                    {
                        write_letter (finfo, 'C');
                        retval = 1;
                    }
                    else
690                 {
                        /* Reregister to clear conflict flag. */
                        Register (finfo->entries, finfo->file, vers->vn_rcs, vers->ts_rcs,
                                  vers->options, vers->tag,
                                  vers->date, (char *)0, CVSroot_directory, finfo->repository);
                    }
                }
                if (!retval)
                {
                    write_letter (finfo, 'M');
700                 retval = 0;
                }
                break;
#ifdef SERVER_SUPPORT
            case T_PATCH:       /* needs patch */
                if (patches)
                {
                    int docheckout;
                    struct stat file_info;
                    unsigned char checksum[16];
710
                    retval = patch_file (finfo,
                                         vers, &docheckout,
                                         &file_info, checksum);
                    if (! docheckout)
                    {
                        if (server_active && retval == 0)
                            server_updated (finfo, vers,
                                            (rcs_diff_patches
```

```
                                        ? SERVER_RCS_DIFF
                                        : SERVER_PATCHED),
                                       file_info.st_mode, checksum,
                                       (struct buffer *) NULL);
                        break;
                    }
                }
                /* If we're not running as a server, just check the
                   file out.  It's simpler and faster than producing
                   and applying patches.  */
                /* Fall through.  */
#endif
            case T_CHECKOUT: /* needs checkout */
                retval = checkout_file (finfo, vers, 0, 0, 1);
                break;
            case T_ADDED:       /* added but not committed */
                write_letter (finfo, 'A');
                retval = 0;
                break;
            case T_REMOVED:   /* removed but not committed */
                write_letter (finfo, 'R');
                retval = 0;
                break;
            case T_REMOVE_ENTRY: /* needs to be un-registered */
                retval = scratch_file (finfo);
#ifdef SERVER_SUPPORT
                if (server_active && retval == 0)
                {
                    if (vers->ts_user == NULL)
                        server_scratch_entry_only ();
                    server_updated (finfo, vers,
                                    SERVER_UPDATED, (mode_t) -1,
                                    (unsigned char *) NULL,
                                    (struct buffer *) NULL);
                }
#endif
                break;

          case T_REMOTE: {
            server_output_not_carried_for_file (finfo, vers);
            retval = 0;
            break;
          }

            default:                /* can't ever happen :-) */
                error (0, 0,
                       "unknown file status %d for file %s", status, finfo->file);
                retval = 0;
                break;
        }
    }

    /* only try to join if things have gone well thus far */
    if (retval == 0 && join_rev1)
        join_file (finfo, vers);

    /* if this directory has an ignore list, add this file to it */
    if (ignlist)
    {
        Node *p;

        p = getnode ();
        p->type = FILES;
        p->key = xstrdup (finfo->file);
        if (addnode (ignlist, p) != 0)
            freenode (p);
    }

    freevers_ts (&vers);
    return (retval);
}

static void update_ignproc PROTO ((char *, char *));

static void
update_ignproc (file, dir)
    char *file;
    char *dir;
{
    struct file_info finfo;

    memset (&finfo, 0, sizeof (finfo));
    finfo.file = file;
    finfo.update_dir = dir;
    if (dir[0] == '\0')
        finfo.fullname = xstrdup (file);
    else
    {
        finfo.fullname = xmalloc (strlen (file) + strlen (dir) + 10);
        strcpy (finfo.fullname, dir);
```

```
        strcat (finfo.fullname, "/");
        strcat (finfo.fullname, file);
    }

    write_letter (&finfo, '?');
    free (finfo.fullname);
}

/* ARGSUSED */
static int
update_filesdone_proc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    if (rewrite_tag)
    {
        WriteTag (NULL, tag, date, nonbranch, update_dir, repository);
        rewrite_tag = 0;
    }

    /* if this directory has an ignore list, process it then free it */
    if (ignlist)
    {
        ignore_files (ignlist, entries, update_dir, update_ignproc);
        dellist (&ignlist);
    }

    /* Clean up CVS admin dirs if we are export */
    if (strcmp (command_name, "export") == 0)
    {
        /* I'm not sure the existence_error is actually possible (except
           in cases where we really should print a message), but since
           this code used to ignore all errors, I'll play it safe.  */
        if (unlink_file_dir (CVSADM) < 0 && !existence_error (errno))
            error (0, errno, "cannot remove %s directory", CVSADM);
    }
#ifdef SERVER_SUPPORT
    else if (!server_active && !pipeout)
#else
    else if (!pipeout)
#endif /* SERVER_SUPPORT */
    {
        /* If there is no CVS/Root file, add one */
        if (!isfile (CVSADM_ROOT))
            Create_Root ((char *) NULL, CVSroot_original);
    }

    return (err);
}

/*
 * update_dirent_proc () is called back by the recursion processor before a
 * sub-directory is processed for update.  In this case, update_dirent proc
 * will probably create the directory unless -d isn't specified and this is a
 * new directory.  A return code of 0 indicates the directory should be
 * processed by the recursion code.  A return of non-zero indicates the
 * recursion code should skip this directory.
 */
static Dtype
update_dirent_proc (callerdat, dir, repository, update_dir, entries)
    void *callerdat;
    char *dir;
    char *repository;
    char *update_dir;
    List *entries;
{
    if (ignore_directory (update_dir))
    {
        /* print the warm fuzzy message */
        if (!quiet)
          error (0, 0, "Ignoring %s", update_dir);
        return R_SKIP_ALL;
    }

    if (!isdir (dir))
    {
        /* if we aren't building dirs, blow it off */
        if (!update_build_dirs)
            return (R_SKIP_ALL);

        if (noexec)
        {
            /* ignore the missing dir if -n is specified */
            error (0, 0, "New directory `%s' -- ignored", update_dir);
            return (R_SKIP_ALL);
        }
        else
```

```
        {
900         /* otherwise, create the dir and appropriate adm files */

            /* If no tag or date were specified on the command line,
               and we're not using -A, we want the subdirectory to use
               the tag and date, if any, of the current directory.
               That way, update -d will work correctly when working on
               a branch.

               We use TAG_UPDATE_DIR to undo the tag setting in
               update_dirleave_proc.  If we did not do this, we would
910            not correctly handle a working directory with multiple
               tags (and maybe we should prohibit such working
               directories, but they work now and we shouldn't make
               them stop working without more thought).  */
            if ((tag == NULL && date == NULL) && ! aflag)
            {
                ParseTag (&tag, &date, &nonbranch);
                if (tag != NULL || date != NULL)
                    tag_update_dir = xstrdup (update_dir);
            }
920
            make_directory (dir);
            Create_Admin (dir, update_dir, repository, tag, date,
                          /* This is a guess.  We will rewrite it later
                             via WriteTag.  */
                          0,
                          0);
            rewrite_tag = 1;
            nonbranch = 0;
            Subdir_Register (entries, (char *) NULL, dir);
930     }
    }
    /* Do we need to check noexec here? */
    else if (!pipeout)
    {
        char *cvsadmdir;

        /* The directory exists.  Check to see if it has a CVS
           subdirectory.  */

940     cvsadmdir = xmalloc (strlen (dir) + 80);
        strcpy (cvsadmdir, dir);
        strcat (cvsadmdir, "/");
        strcat (cvsadmdir, CVSADM);

        if (!isdir (cvsadmdir))
        {
            /* We cannot successfully recurse into a directory without a CVS
               subdirectory.  Generally we will have already printed
               "? foo".  */
950         free (cvsadmdir);
            return R_SKIP_ALL;
        }
        free (cvsadmdir);
    }

    /*
     * If we are building dirs and not going to stdout, we make sure there is
     * no static entries file and write the tag file as appropriate
     */
960 if (!pipeout)
    {
        if (update_build_dirs)
        {
            char *tmp;

            tmp = xmalloc (strlen (dir) + sizeof (CVSADM_ENTSTAT) + 10);
            (void) sprintf (tmp, "%s/%s", dir, CVSADM_ENTSTAT);
            if (unlink_file (tmp) < 0 && ! existence_error (errno))
                error (1, errno, "cannot remove file %s", tmp);
970 #ifdef SERVER_SUPPORT
            if (server_active)
                server_clear_entstat (update_dir, repository);
#endif
            free (tmp);
        }

        /* keep the CVS/Tag file current with the specified arguments */
        if (aflag || tag || date)
        {
980         WriteTag (dir, tag, date, 0, update_dir, repository);
            rewrite_tag = 1;
            nonbranch = 0;
        }

        /* initialize the ignore list for this directory */
        ignlist = getlist ();
    }
```

```
    /* print the warm fuzzy message */
990 if (!quiet)
        error (0, 0, "Updating %s", update_dir);

    return (R_PROCESS);
}

/*
 * update_dirleave_proc () is called back by the recursion code upon leaving
 * a directory.  It will prune empty directories if needed and will execute
 * any appropriate update programs.
1000 */
/* ARGSUSED */
static int
update_dirleave_proc (callerdat, dir, err, update_dir, entries)
    void *callerdat;
    char *dir;
    int err;
    char *update_dir;
    List *entries;
{
1010 FILE *fp;

    /* If we set the tag or date for a new subdirectory in
       update_dirent_proc, and we're now done with that subdirectory,
       undo the tag/date setting.  Note that we know that the tag and
       date were both originally NULL in this case.  */
    if (tag_update_dir != NULL && strcmp (update_dir, tag_update_dir) == 0)
    {
        if (tag != NULL)
        {
1020        free (tag);
            tag = NULL;
        }
        if (date != NULL)
        {
            free (date);
            date = NULL;
        }
        nonbranch = 0;
        free (tag_update_dir);
1030    tag_update_dir = NULL;
    }

    /* run the update_prog if there is one */
    /* FIXME: should be checking for errors from CVS_FOPEN and printing
       them if not existence_error.  */
    if (err == 0 && !pipeout && !noexec &&
        (fp = CVS_FOPEN (CVSADM_UPROG, "r")) != NULL)
    {
        char *cp;
1040    char *repository;
        char *line = NULL;
        size_t line_allocated = 0;

        repository = Name_Repository ((char *) NULL, update_dir);
        if (getline (&line, &line_allocated, fp) >= 0)
        {
            if ((cp = strrchr (line, '\n')) != NULL)
                *cp = '\0';
            run_setup (line);
1050        run_arg (repository);
            cvs_output (program_name, 0);
            cvs_output (" ", 1);
            cvs_output (command_name, 0);
            cvs_output (": Executing '", 0);
            run_print (stdout);
            cvs_output ("'\n", 0);
            (void) run_exec (RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL);
        }
        else if (ferror (fp))
1060        error (0, errno, "cannot read %s", CVSADM_UPROG);
        else
            error (0, 0, "unexpected end of file on %s", CVSADM_UPROG);

        if (fclose (fp) < 0)
            error (0, errno, "cannot close %s", CVSADM_UPROG);
        if (line != NULL)
            free (line);
        free (repository);
    }
1070
    if (strchr (dir, '/') == NULL)
    {
        /* FIXME: chdir (".") loses with symlinks.  */
        /* Prune empty dirs on the way out - if necessary */
        (void) CVS_CHDIR ("..");
        if (update_prune_dirs && isemptydir (dir, 0))
        {
            /* I'm not sure the existence_error is actually possible (except
```

```
            in cases where we really should print a message), but since
            this code used to ignore all errors, I'll play it safe. */
          if (unlink_file_dir (dir) < 0 && !existence_error (errno))
              error (0, errno, "cannot remove %s directory", dir);
          Subdir_Deregister (entries, (char *) NULL, dir);
        }
    }

    return (err);
}

static int isremoved PROTO ((Node *, void *));

/* Returns 1 if the file indicated by node has been removed.  */
static int
isremoved (node, closure)
    Node *node;
    void *closure;
{
    Entnode *entdata = (Entnode*) node->data;

    /* If the first character of the version is a '-', the file has been
       removed. */
    return (entdata->version && entdata->version[0] == '-') ? 1 : 0;
}

/* Returns 1 if the argument directory is completely empty, other than the
   existence of the CVS directory entry.  Zero otherwise.  If MIGHT_NOT_EXIST
   and the directory doesn't exist, then just return 0.  */
int
isemptydir (dir, might_not_exist)
    char *dir;
    int might_not_exist;
{
    DIR *dirp;
    struct dirent *dp;

    if ((dirp = CVS_OPENDIR (dir)) == NULL)
    {
        if (might_not_exist && existence_error (errno))
            return 0;
        error (0, errno, "cannot open directory %s for empty check", dir);
        return (0);
    }
    errno = 0;
    while ((dp = readdir (dirp)) != NULL)
    {
        if (strcmp (dp->d_name, ".") != 0
            && strcmp (dp->d_name, "..") != 0)
        {
            if (strcmp (dp->d_name, CVSADM) != 0)
            {
                /* An entry other than the CVS directory.  The directory
                   is certainly not empty. */
                (void) closedir (dirp);
                return (0);
            }
            else
            {
                /* The CVS directory entry.  We don't have to worry about
                   this unless the Entries file indicates that files have
                   been removed, but not committed, in this directory.
                   (Removing the directory would prevent people from
                   comitting the fact that they removed the files!) */
                List *l;
                int files_removed;
                struct saved_cwd cwd;

                if (save_cwd (&cwd))
                    error_exit ();

                if (CVS_CHDIR (dir) < 0)
                    error (1, errno, "cannot change directory to %s", dir);
                l = Entries_Open (0, NULL);
                files_removed = walklist (l, isremoved, 0);
                Entries_Close (l);

                if (restore_cwd (&cwd, NULL))
                    error_exit ();
                free_cwd (&cwd);

                if (files_removed != 0)
                {
                    /* There are files that have been removed, but not
                       committed!  Do not consider the directory empty. */
                    (void) closedir (dirp);
                    return (0);
                }
            }
        }
```

```
            errno = 0;
        }
        if (errno != 0)
        {
            error (0, errno, "cannot read directory %s", dir);
            (void) closedir (dirp);
            return (0);
        }
        (void) closedir (dirp);
        return (1);
    }

    /*
     * scratch the Entries file entry associated with a file
     */
    static int
    scratch_file (finfo)
        struct file_info *finfo;
    {
        history_write ('W', finfo->update_dir, "", finfo->file, finfo->repository);
        Scratch_Entry (finfo->entries, finfo->file);
        if (unlink_file (finfo->file) < 0 && ! existence_error (errno))
            error (0, errno, "unable to remove %s", finfo->fullname);
        return (0);
    }

    /*
     * Check out a file.
     */
    static int
    checkout_file (finfo, vers_ts, adding, merging, update_server)
        struct file_info *finfo;
        Vers_TS *vers_ts;
        int adding;
        int merging;
        int update_server;
    {
        char *backup;
        int set_time, retval = 0;
        int status;
        int file_is_dead;
        struct buffer *revbuf;

        backup = NULL;
        revbuf = NULL;

        /* Don't screw with backup files if we're going to stdout, or if
           we are the server.  */
        if (!pipeout
    #ifdef SERVER_SUPPORT
            && ! server_active
    #endif
            )
        {
            backup = xmalloc (strlen (finfo->file)
                              + sizeof (CVSADM)
                              + sizeof (CVSPREFIX)
                              + 10);
            (void) sprintf (backup, "%s/%s%s", CVSADM, CVSPREFIX, finfo->file);
            if (isfile (finfo->file))
                rename_file (finfo->file, backup);
            else
            {
                /* If -f/-t wrappers are being used to wrap up a directory,
                   then backup might be a directory instead of just a file.  */
                if (unlink_file_dir (backup) < 0)
                {
                    /* Not sure if the existence_error check is needed here.  */
                    if (!existence_error (errno))
                        /* FIXME: should include update_dir in message.  */
                        error (0, errno, "error removing %s", backup);
                }
                free (backup);
                backup = NULL;
            }
        }

        file_is_dead = RCS_isdead (vers_ts->srcfile, vers_ts->vn_rcs);

        if (!file_is_dead)
        {
            /*
             * if we are checking out to stdout, print a nice message to
             * stderr, and add the -p flag to the command */
            if (pipeout)
            {
                if (!quiet)
                {
                    cvs_outerr ("\
===================================================================\n\
```

```
        Checking out ", 0);
1260                cvs_outerr (finfo->fullname, 0);
                    cvs_outerr ("\n\
        RCS:  ", 0);
                    cvs_outerr (vers_ts->srcfile->path, 0);
                    cvs_outerr ("\n\
        VERS: ", 0);
                    cvs_outerr (vers_ts->vn_rcs, 0);
                    cvs_outerr ("\n***************\n", 0);
                }
            }
1270
        #ifdef SERVER_SUPPORT
            if (update_server
                && server_active
                && ! pipeout
                && ! file_gzip_level
                && ! joining ()
                && ! wrap_name_has (finfo->file, WRAP_FROMCVS))
            {
                revbuf = buf_nonio_initialize ((BUFMEMERRPROC) NULL);
1280            status = RCS_checkout (vers_ts->srcfile, (char *) NULL,
                                       vers_ts->vn_rcs, vers_ts->vn_tag,
                                       vers_ts->options, RUN_TTY,
                                       checkout_to_buffer, revbuf);
            }
            else
        #endif
        #ifdef CLIENT_SUPPORT
             if (client_active && handling_remotes && fetch_remote (finfo)) {
               /* If we are the client, and this checkout is a part of
1290            * fetching a remote revision for the purpose of feeding
                * it back to a server, then we check it out to a different
                * location (inside CVS/). See client.c.
                */
               int len;
               char* real_file = NULL;
               char* last_slash;

               len = strlen (finfo -> file) + strlen (CVSADM) + strlen (vers_ts -> vn_tag) +
                 strlen (CVSADM_CACHED_REMOTE_SEPARATOR) + 1;
1300           real_file = xmalloc (len);
               if (real_file == NULL) {
                 error (1, errno, "checkout failed");
               }

               last_slash = strrchr (finfo -> file, '/');
               if (last_slash == NULL) {
                 last_slash = finfo -> file;
               } else {
                 last_slash++;
1310           }
               sprintf (real_file, "%.*s%s%s%s%s", finfo -> file, last_slash - finfo -> file,
                        CVSADM, last_slash, CVSADM_CACHED_REMOTE_SEPARATOR, vers_ts -> vn_tag);

               printf ("Checking out to %s instead of %s.\n", real_file, finfo -> file);

               status = RCS_checkout (vers_ts -> srcfile, real_file, vers_ts -> vn_rcs,
                                      vers_ts -> vn_tag, vers_ts -> options, RUN_TTY, NULL, NULL);
               if (status == 0) {
                 FILE* remotes_file = fopen (CVSADM_REMOTES, "a+");
1320             if (remotes_file != NULL) {
                   fprintf (remotes_file, "%s/%s/%s\n", finfo -> file, vers_ts -> vn_rcs, strchr (real_file, '/') + 1);
                   fclose (remotes_file);
                 }
               }

             } else
        #endif
               status = RCS_checkout (vers_ts->srcfile,
                                      pipeout ? NULL : finfo->file,
1330                                  vers_ts->vn_rcs, vers_ts->vn_tag,
                                      vers_ts->options, RUN_TTY,
                                      (RCSCHECKOUTPROC) NULL, (void *) NULL);
        }

        if (file_is_dead || status == 0)
        {
            mode_t mode;

            mode = (mode_t) -1;
1340
            if (!pipeout)
            {
                Vers_TS *xvers_ts;

                if (revbuf != NULL)
                {
                    struct stat sb;
```

```
            /* FIXME: We should have RCS_checkout return the mode.  */
            if (stat (vers_ts->srcfile->path, &sb) < 0)
                error (1, errno, "cannot stat %s",
                       vers_ts->srcfile->path);
            mode = sb.st_mode &~ (S_IWRITE | S_IWGRP | S_IWOTH);
        }

        if (cvswrite
            && !file_is_dead
            && !fileattr_get (finfo->file, "_watched"))
        {
            if (revbuf == NULL)
                xchmod (finfo->file, 1);
            else
            {
                /* We know that we are the server here, so
                   although xchmod checks umask, we don't bother.  */
                mode |= (((mode & S_IRUSR) ? S_IWUSR : 0)
                         | ((mode & S_IRGRP) ? S_IWGRP : 0)
                         | ((mode & S_IROTH) ? S_IWOTH : 0));
            }
        }

        {
            /* A newly checked out file is never under the spell
               of "cvs edit".  If we think we were editing it
               from a previous life, clean up.  Would be better to
               check for same the working directory instead of
               same user, but that is hairy.  */

            struct addremove_args args;

            editor_set (finfo->file, getcaller (), NULL);

            memset (&args, 0, sizeof args);
            args.remove_temp = 1;
            watch_modify_watchers (finfo->file, &args);
        }

        /* set the time from the RCS file iff it was unknown before */
        set_time =
            (!noexec
             && (vers_ts->vn_user == NULL ||
                 strncmp (vers_ts->ts_rcs, "Initial", 7) == 0)
             && !file_is_dead);

        wrap_fromcvs_process_file (finfo->file);

        xvers_ts = Version_TS (finfo, options, tag, date,
                               force_tag_match, set_time);
        if (strcmp (xvers_ts->options, "-V4") == 0)
            xvers_ts->options[0] = '\0';

        if (revbuf != NULL)
        {
            /* If we stored the file data into a buffer, then we
               didn't create a file at all, so xvers_ts->ts_user
               is wrong.  The correct value is to have it be the
               same as xvers_ts->ts_rcs, meaning that the working
               file is unchanged from the RCS file.

               FIXME: We should tell Version_TS not to waste time
               statting the nonexistent file.

               FIXME: Actually, I don't think the ts_user value
               matters at all here.  The only use I know of is
               that it is printed in a trace message by
               Server_Register.  */

            if (xvers_ts->ts_user != NULL)
                free (xvers_ts->ts_user);
            xvers_ts->ts_user = xstrdup (xvers_ts->ts_rcs);
        }

        (void) time (&last_register_time);

        if (file_is_dead)
        {
            if (xvers_ts->vn_user != NULL)
            {
                error (0, 0,
                       "warning: %s is not (any longer) pertinent",
                       finfo->fullname);
            }
            Scratch_Entry (finfo->entries, finfo->file);
#ifdef SERVER_SUPPORT
            if (server_active && xvers_ts->ts_user == NULL)
                server_scratch_entry_only ();
#endif
            /* FIXME: Rather than always unlink'ing, and ignoring the
```

```
                existence_error, we should do the unlink only if
                vers_ts->ts_user is non-NULL.  Then there would be no
                need to ignore an existence_error (for example, if the
                user removes the file while we are running).  */
             if (unlink_file (finfo->file) < 0 && ! existence_error (errno))
             {
                error (0, errno, "cannot remove %s", finfo->fullname);
             }
          }
          else {
            /* Since this is a fresh checkout, we give it the appropriate repository string */
            /* meeroh fixme: add the root, not just the repositoy location */
             Register (finfo->entries, finfo->file,
                       adding ? "0" : xvers_ts->vn_rcs,
                       xvers_ts->ts_user, xvers_ts->options,
                       xvers_ts->tag, xvers_ts->date,
                       (char *)0, CVSroot_original, CVSroot_directory); /* Clear conflict flag on fresh checkout */
          }

          /* fix up the vers structure, in case it is used by join */
          if (join_rev1)
          {
             if (vers_ts->vn_user != NULL)
                free (vers_ts->vn_user);
             if (vers_ts->vn_rcs != NULL)
                free (vers_ts->vn_rcs);
             vers_ts->vn_user = xstrdup (xvers_ts->vn_rcs);
             vers_ts->vn_rcs = xstrdup (xvers_ts->vn_rcs);
          }

          /* If this is really Update and not Checkout, recode history */
          if (strcmp (command_name, "update") == 0)
             history_write ('U', finfo->update_dir, xvers_ts->vn_rcs, finfo->file,
                            finfo->repository);

          freevers_ts (&xvers_ts);

          if (!really_quiet && !file_is_dead)
          {
             write_letter (finfo, 'U');
          }
       }

#ifdef SERVER_SUPPORT
       if (update_server && server_active)
          server_updated (finfo, vers_ts,
                          merging ? SERVER_MERGED : SERVER_UPDATED,
                          mode, (unsigned char *) NULL, revbuf);
#endif
    }
    else
    {
       if (backup != NULL)
       {
          rename_file (backup, finfo->file);
          free (backup);
          backup = NULL;
       }

       error (0, 0, "could not check out %s", finfo->fullname);

       retval = status;
    }

    if (backup != NULL)
    {
       /* If -f/-t wrappers are being used to wrap up a directory,
          then backup might be a directory instead of just a file.  */
       if (unlink_file_dir (backup) < 0)
       {
          /* Not sure if the existence_error check is needed here.  */
          if (!existence_error (errno))
             /* FIXME: should include update_dir in message.  */
             error (0, errno, "error removing %s", backup);
       }
       free (backup);
    }

    return (retval);
}

#ifdef SERVER_SUPPORT

/* This function is used to write data from a file being checked out
   into a buffer.  */

static void
checkout_to_buffer (callerdat, data, len)
     void *callerdat;
     const char *data;
```

```
    size_t len;
{
    struct buffer *buf = (struct buffer *) callerdat;

    buf_output (buf, data, len);
}

#endif /* SERVER_SUPPORT */

#ifdef SERVER_SUPPORT

/* This structure is used to pass information between patch_file and
   patch_file_write.  */

struct patch_file_data
{
    /* File name, for error messages.  */
    const char *filename;
    /* File to which to write.  */
    FILE *fp;
    /* Whether to compute the MD5 checksum.  */
    int compute_checksum;
    /* Data structure for computing the MD5 checksum.  */
    struct MD5Context context;
    /* Set if the file has a final newline.  */
    int final_nl;
};

/* Patch a file.  Runs diff.  This is only done when running as the
 * server.  The hope is that the diff will be smaller than the file
 * itself.
 */
static int
patch_file (finfo, vers_ts, docheckout, file_info, checksum)
    struct file_info *finfo;
    Vers_TS *vers_ts;
    int *docheckout;
    struct stat *file_info;
    unsigned char *checksum;
{
    char *backup;
    char *file1;
    char *file2;
    int retval = 0;
    int retcode = 0;
    int fail;
    FILE *e;
    struct patch_file_data data;

    *docheckout = 0;

    if (noexec || pipeout || joining ())
    {
        *docheckout = 1;
        return 0;
    }

    /* If this file has been marked as being binary, then never send a
       patch.  */
    if (strcmp (vers_ts->options, "-kb") == 0)
    {
        *docheckout = 1;
        return 0;
    }

    /* First check that the first revision exists.  If it has been nuked
       by cvs admin -o, then just fall back to checking out entire
       revisions.  In some sense maybe we don't have to do this; after
       all cvs.texinfo says "Make sure that no-one has checked out a
       copy of the revision you outdate" but then again, that advice
       doesn't really make complete sense, because "cvs admin" operates
       on a working directory and so _someone_ will almost always have
       _some_ revision checked out.  */
    {
        char *rev;

        rev = RCS_gettag (finfo->rcs, vers_ts->vn_user, 1, NULL);
        if (rev == NULL)
        {
            *docheckout = 1;
            return 0;
        }
        else
            free (rev);
    }

    /* If the revision is dead, let checkout_file handle it rather
       than duplicating the processing here.  */
    if (RCS_isdead (vers_ts->srcfile, vers_ts->vn_rcs))
    {
```

```
        *docheckout = 1;
        return 0;
    }

    backup = xmalloc (strlen (finfo->file)
                      + sizeof (CVSADM)
                      + sizeof (CVSPREFIX)
                      + 10);
    (void) sprintf (backup, "%s/%s%s", CVSADM, CVSPREFIX, finfo->file);
    if (isfile (finfo->file))
        rename_file (finfo->file, backup);
    else
        (void) unlink_file (backup);

    file1 = xmalloc (strlen (finfo->file)
                     + sizeof (CVSADM)
                     + sizeof (CVSPREFIX)
                     + 10);
    (void) sprintf (file1, "%s/%s%s-1", CVSADM, CVSPREFIX, finfo->file);
    file2 = xmalloc (strlen (finfo->file)
                     + sizeof (CVSADM)
                     + sizeof (CVSPREFIX)
                     + 10);
    (void) sprintf (file2, "%s/%s%s-2", CVSADM, CVSPREFIX, finfo->file);

    fail = 0;

    /* We need to check out both revisions first, to see if either one
       has a trailing newline.  Because of this, we don't use rcsdiff,
       but just use diff.  */

    e = CVS_FOPEN (file1, "w");
    if (e == NULL)
        error (1, errno, "cannot open %s", file1);

    data.filename = file1;
    data.fp = e;
    data.final_nl = 0;
    data.compute_checksum = 0;

    retcode = RCS_checkout (vers_ts->srcfile, (char *) NULL,
                            vers_ts->vn_user, (char *) NULL,
                            vers_ts->options, RUN_TTY,
                            patch_file_write, (void *) &data);

    if (fclose (e) < 0)
        error (1, errno, "cannot close %s", file1);

    if (retcode != 0 || ! data.final_nl)
        fail = 1;

    if (! fail)
    {
        e = CVS_FOPEN (file2, "w");
        if (e == NULL)
            error (1, errno, "cannot open %s", file2);

        data.filename = file2;
        data.fp = e;
        data.final_nl = 0;
        data.compute_checksum = 1;
        MD5Init (&data.context);

        retcode = RCS_checkout (vers_ts->srcfile, (char *) NULL,
                                vers_ts->vn_rcs, (char *) NULL,
                                vers_ts->options, RUN_TTY,
                                patch_file_write, (void *) &data);

        if (fclose (e) < 0)
            error (1, errno, "cannot close %s", file2);

        if (retcode != 0 || ! data.final_nl)
            fail = 1;
        else
            MD5Final (checksum, &data.context);
    }

    retcode = 0;
    if (! fail)
    {
        char *diff_options;

        /* If the client does not support the Rcs-diff command, we
           send a context diff, and the client must invoke patch.
           That approach was problematical for various reasons.  The
           new approach only requires running diff in the server; the
           client can handle everything without invoking an external
           program.  */
        if (! rcs_diff_patches)
        {
```

```
            /* We use -c, not -u, because that is what CVS has
               traditionally used.  Kind of a moot point, now that
               Rcs-diff is preferred, so there is no point in making
               the compatibility issues worse.  */
            diff_options = "-c";
        }
        else
        {
            /* Now that diff is librarified, we could be passing -a if
               we wanted to.  However, it is unclear to me whether we
               would want to.  Does diff -a, in any significant
               percentage of cases, produce patches which are smaller
               than the files it is patching?  I guess maybe text
               files with character sets which diff regards as
               'binary'.  Conversely, do they tend to be much larger
               in the bad cases?  This needs some more
               thought/investigation, I suspect.  */

            diff_options = "-n";
        }
        retcode = diff_exec (file1, file2, diff_options, finfo->file);

        /* A retcode of 0 means no differences.  1 means some differences.  */
        if (retcode != 0
            && retcode != 1)
        {
            fail = 1;
        }
        else
        {
#define BINARY "Binary"
            char buf[sizeof BINARY];
            unsigned int c;

            /* Stat the original RCS file, and then adjust it the way
               that RCS_checkout would.  FIXME: This is an abstraction
               violation.  */
            if (CVS_STAT (vers_ts->srcfile->path, file_info) < 0)
                error (1, errno, "could not stat %s", vers_ts->srcfile->path);
            if (chmod (finfo->file,
                       file_info->st_mode & ~(S_IWRITE | S_IWGRP | S_IWOTH))
                < 0)
                error (0, errno, "cannot change mode of file %s", finfo->file);
            if (cvswrite
                && !fileattr_get (finfo->file, "_watched"))
                xchmod (finfo->file, 1);

            /* Check the diff output to make sure patch will be handle it.  */
            e = CVS_FOPEN (finfo->file, "r");
            if (e == NULL)
                error (1, errno, "could not open diff output file %s",
                       finfo->fullname);
            c = fread (buf, 1, sizeof BINARY - 1, e);
            buf[c] = '\0';
            if (strcmp (buf, BINARY) == 0)
            {
                /* These are binary files.  We could use diff -a, but
                   patch can't handle that.  */
                fail = 1;
            }
            fclose (e);
        }
    }

    if (! fail)
    {
        Vers_TS *xvers_ts;

        /* This stuff is just copied blindly from checkout_file.  I
           don't really know what it does.  */
        xvers_ts = Version_TS (finfo, options, tag, date,
                               force_tag_match, 0);
        if (strcmp (xvers_ts->options, "-V4") == 0)
            xvers_ts->options[0] = '\0';

        Register (finfo->entries, finfo->file, xvers_ts->vn_rcs,
                  xvers_ts->ts_user, xvers_ts->options,
                  xvers_ts->tag, xvers_ts->date, NULL, CVSroot_directory, finfo->repository);

        if (CVS_STAT (finfo->file, file_info) < 0)
            error (1, errno, "could not stat %s", finfo->file);

        /* If this is really Update and not Checkout, recode history */
        if (strcmp (command_name, "update") == 0)
            history_write ('P', finfo->update_dir, xvers_ts->vn_rcs, finfo->file,
                           finfo->repository);

        freevers_ts (&xvers_ts);

        if (!really_quiet)
```

```
        {
1800        write_letter (finfo, 'P');
        }
    }
    else
    {
        int old_errno = errno;    /* save errno value over the rename */

        if (isfile (backup))
            rename_file (backup, finfo->file);

1810        if (retcode != 0 && retcode != 1)
            error (retcode == -1 ? 1 : 0, retcode == -1 ? old_errno : 0,
                   "could not diff %s", finfo->fullname);

        *docheckout = 1;
        retval = retcode;
    }

    (void) unlink_file (backup);
    (void) unlink_file (file1);
1820    (void) unlink_file (file2);

    free (backup);
    free (file1);
    free (file2);
    return (retval);
}

/* Write data to a file.  Record whether the last byte written was a
   newline.  Optionally compute a checksum.  This is called by
1830   patch_file via RCS_checkout.  */

static void
patch_file_write (callerdat, buffer, len)
     void *callerdat;
     const char *buffer;
     size_t len;
{
    struct patch_file_data *data = (struct patch_file_data *) callerdat;

1840    if (fwrite (buffer, 1, len, data->fp) != len)
        error (1, errno, "cannot write %s", data->filename);

    data->final_nl = (buffer[len - 1] == '\n');

    if (data->compute_checksum)
        MD5Update (&data->context, (unsigned char *) buffer, len);
}

#endif /* SERVER_SUPPORT */
1850
/*
 * Several of the types we process only print a bit of information consisting
 * of a single letter and the name.
 */
static void
write_letter (finfo, letter)
    struct file_info *finfo;
    int letter;
{
1860    if (!really_quiet)
    {
        char *tag = NULL;
        /* Big enough for "+updated" or any of its ilk.  */
        char buf[80];

        switch (letter)
        {
            case 'U':
                tag = "updated";
1870            break;
            default:
                /* We don't yet support tagged output except for "U".  */
                break;
        }

        if (tag != NULL)
        {
            sprintf (buf, "+%s", tag);
            cvs_output_tagged (buf, NULL);
1880        }
        buf[0] = letter;
        buf[1] = ' ';
        buf[2] = '\0';
        cvs_output_tagged ("text", buf);
        cvs_output_tagged ("fname", finfo->fullname);
        cvs_output_tagged ("newline", NULL);
        if (tag != NULL)
        {
```

```
            sprintf (buf, "-%s", tag);
            cvs_output_tagged (buf, NULL);
        }
    }
    return;
}

/*
 * Do all the magic associated with a file which needs to be merged
 */
static int
merge_file (finfo, vers)
    struct file_info *finfo;
    Vers_TS *vers;
{
    char *backup;
    int status;
    int retcode = 0;
    int retval;

    /*
     * The users currently modified file is moved to a backup file name
     * ".#filename.version", so that it will stay around for a few days
     * before being automatically removed by some cron daemon.  The "version"
     * is the version of the file that the user was most up-to-date with
     * before the merge.
     */
    backup = xmalloc (strlen (finfo->file)
                      + strlen (vers->vn_user)
                      + sizeof (BAKPREFIX)
                      + 10);
    (void) sprintf (backup, "%s%s.%s", BAKPREFIX, finfo->file, vers->vn_user);

    (void) unlink_file (backup);
    copy_file (finfo->file, backup);
    xchmod (finfo->file, 1);

    if (strcmp (vers->options, "-kb") == 0
        || wrap_merge_is_copy (finfo->file)
        || special_file_mismatch (finfo, NULL, vers->vn_rcs))
    {
        /* For binary files, a merge is always a conflict.  Same for
           files whose permissions or linkage do not match.  We give the
           user the two files, and let them resolve it.  It is possible
           that we should require a "touch foo" or similar step before
           we allow a checkin.  */

        /* TODO: it may not always be necessary to regard a permission
           mismatch as a conflict.  The working file and the RCS file
           have a common ancestor 'A'; if the working file's permissions
           match A's, then it's probably safe to overwrite them with the
           RCS permissions.  Only if the working file, the RCS file, and
           A all disagree should this be considered a conflict.  But more
           thought needs to go into this, and in the meantime it is safe
           to treat any such mismatch as an automatic conflict.  -twp */

#ifdef SERVER_SUPPORT
        if (server_active)
            server_copy_file (finfo->file, finfo->update_dir,
                              finfo->repository, backup);
#endif

        status = checkout_file (finfo, vers, 0, 1, 1);

        /* Is there a better term than "nonmergeable file"?  What we
           really mean is, not something that CVS cannot or does not
           want to merge (there might be an external manual or
           automatic merge process).  */
        error (0, 0, "nonmergeable file needs merge");
        error (0, 0, "revision %s from repository is now in %s",
               vers->vn_rcs, finfo->fullname);
        error (0, 0, "file from working directory is now in %s", backup);
        write_letter (finfo, 'C');

        history_write ('C', finfo->update_dir, vers->vn_rcs, finfo->file,
                       finfo->repository);
        retval = 0;
        goto out;
    }

    status = RCS_merge(finfo->rcs, vers->srcfile->path, finfo->file,
                       vers->options, vers->vn_user, vers->vn_rcs);
    if (status != 0 && status != 1)
    {
        error (0, status == -1 ? errno : 0,
               "could not merge revision %s of %s", vers->vn_user, finfo->fullname);
        error (status == -1 ? 1 : 0, 0, "restoring %s from backup file %s",
               finfo->fullname, backup);
        rename_file (backup, finfo->file);
        retval = 1;
```

```
            goto out;
1980    }

        if (strcmp (vers->options, "-V4") == 0)
            vers->options[0] = '\0';

        /* This file is the result of a merge, which means that it has
           been modified.  We use a special timestamp string which will
           not compare equal to any actual timestamp.  */
        {
            char *cp = 0;
1990
            if (status)
            {
                (void) time (&last_register_time);
                cp = time_stamp (finfo->file);
            }
            Register (finfo->entries, finfo->file, vers->vn_rcs,
                      "Result of merge", vers->options, vers->tag,
                      vers->date, cp, CVSroot_directory, finfo->repository);
            if (cp)
2000            free (cp);
        }

        /* fix up the vers structure, in case it is used by join */
        if (join_rev1)
        {
            if (vers->vn_user != NULL)
                free (vers->vn_user);
            vers->vn_user = xstrdup (vers->vn_rcs);
        }
2010
    #ifdef SERVER_SUPPORT
        /* Send the new contents of the file before the message.  If we
           wanted to be totally correct, we would have the client write
           the message only after the file has safely been written.  */
        if (server_active)
        {
            server_copy_file (finfo->file, finfo->update_dir, finfo->repository,
                              backup);
            server_updated (finfo, vers, SERVER_MERGED,
2020                        (mode_t) -1, (unsigned char *) NULL,
                            (struct buffer *) NULL);
        }
    #endif

        if (!noexec && !xcmp (backup, finfo->file))
        {
            printf ("%s already contains the differences between %s and %s\n",
                    finfo->fullname, vers->vn_user, vers->vn_rcs);
            history_write ('G', finfo->update_dir, vers->vn_rcs, finfo->file,
2030                       finfo->repository);
            retval = 0;
            goto out;
        }

        if (status == 1)
        {
            if (!noexec)
                error (0, 0, "conflicts found in %s", finfo->fullname);

2040        write_letter (finfo, 'C');

            history_write ('C', finfo->update_dir, vers->vn_rcs, finfo->file, finfo->repository);

        }
        else if (retcode == -1)
        {
            error (1, errno, "fork failed while examining update of %s",
                   finfo->fullname);
        }
2050    else
        {
            write_letter (finfo, 'M');
            history_write ('G', finfo->update_dir, vers->vn_rcs, finfo->file,
                           finfo->repository);
        }
        retval = 0;
     out:
        free (backup);
        return retval;
2060 }

    /*
     * Do all the magic associated with a file which needs to be joined
     * (-j option)
     */
    static void
    join_file (finfo, vers)
        struct file_info *finfo;
```

```
    Vers_TS *vers;
{
    char *backup;
    char *options;
    int status;

    char *rev1;
    char *rev2;
    char *jrev1;
    char *jrev2;
    char *jdate1;
    char *jdate2;

    jrev1 = join_rev1;
    jrev2 = join_rev2;
    jdate1 = date_rev1;
    jdate2 = date_rev2;

    /* Determine if we need to do anything at all.  */
    if (vers->srcfile == NULL ||
        vers->srcfile->path == NULL)
    {
        return;
    }

    /* If only one join revision is specified, it becomes the second
       revision.  */
    if (jrev2 == NULL)
    {
        jrev2 = jrev1;
        jrev1 = NULL;
        jdate2 = jdate1;
        jdate1 = NULL;
    }

    /* Convert the second revision, walking branches and dates.  */
    rev2 = RCS_getversion (vers->srcfile, jrev2, jdate2, 1, (int *) NULL);

    /* If this is a merge of two revisions, get the first revision.
       If only one join tag was specified, then the first revision is
       the greatest common ancestor of the second revision and the
       working file.  */
    if (jrev1 != NULL)
        rev1 = RCS_getversion (vers->srcfile, jrev1, jdate1, 1, (int *) NULL);
    else
    {
        /* Note that we use vn_rcs here, since vn_user may contain a
           special string such as "-nn".  */
        if (vers->vn_rcs == NULL)
            rev1 = NULL;
        else if (rev2 == NULL)
        {
            /* This means that the file never existed on the branch.
               It does not mean that the file was removed on the
               branch: that case is represented by a dead rev2.  If
               the file never existed on the branch, then we have
               nothing to merge, so we just return.  */
            return;
        }
        else
            rev1 = gca (vers->vn_rcs, rev2);
    }

    /* Handle a nonexistent or dead merge target.  */
    if (rev2 == NULL || RCS_isdead (vers->srcfile, rev2))
    {
        char *mrev;

        if (rev2 != NULL)
            free (rev2);

        /* If the first revision doesn't exist either, then there is
           no change between the two revisions, so we don't do
           anything.  */
        if (rev1 == NULL || RCS_isdead (vers->srcfile, rev1))
        {
            if (rev1 != NULL)
                free (rev1);
            return;
        }

        /* If we are merging two revisions, then the file was removed
           between the first revision and the second one.  In this
           case we want to mark the file for removal.

           If we are merging one revision, then the file has been
           removed between the greatest common ancestor and the merge
           revision.  From the perspective of the branch on to which
           we ar emerging, which may be the trunk, either 1) the file
           does not currently exist on the target, or 2) the file has
```

```
   not been modified on the target branch since the greatest
2160 common ancestor, or 3) the file has been modified on the
   target branch since the greatest common ancestor. In case
   1 there is nothing to do. In case 2 we mark the file for
   removal. In case 3 we have a conflict.

   Note that the handling is slightly different depending upon
   whether one or two join targets were specified. If two
   join targets were specified, we don't check whether the
   file was modified since a given point. My reasoning is
   that if you ask for an explicit merge between two tags,
2170 then you want to merge in whatever was changed between
   those two tags. If a file was removed between the two
   tags, then you want it to be removed. However, if you ask
   for a merge of a branch, then you want to merge in all
   changes which were made on the branch. If a file was
   removed on the branch, that is a change to the file. If
   the file was also changed on the main line, then that is
   also a change. These two changes–the file removal and the
   modification–must be merged. This is a conflict. */

2180 /* If the user file is dead, or does not exist, or has been
   marked for removal, then there is nothing to do. */
if (vers->vn_user == NULL
    || vers->vn_user[0] == '-'
    || RCS_isdead (vers->srcfile, vers->vn_user))
{
    if (rev1 != NULL)
        free (rev1);
    return;
}
2190
/* If the user file has been marked for addition, or has been
   locally modified, then we have a conflict which we can not
   resolve. No_Difference will already have been called in
   this case, so comparing the timestamps is sufficient to
   determine whether the file is locally modified. */
if (strcmp (vers->vn_user, "0") == 0
    || (vers->ts_user != NULL
        && strcmp (vers->ts_user, vers->ts_rcs) != 0))
{
2200    if (jdate2 != NULL)
        error (0, 0,
               "file %s is locally modified, but has been removed in revision %s as of %s",
               finfo->fullname, jrev2, jdate2);
    else
        error (0, 0,
               "file %s is locally modified, but has been removed in revision %s",
               finfo->fullname, jrev2);

    /* FIXME: Should we arrange to return a non-zero exit
2210   status? */

    if (rev1 != NULL)
        free (rev1);

    return;
}

/* If only one join tag was specified, and the user file has
   been changed since the greatest common ancestor (rev1),
2220 then there is a conflict we can not resolve. See above for
   the rationale. */
if (join_rev2 == NULL
    && strcmp (rev1, vers->vn_user) != 0)
{
    if (jdate2 != NULL)
        error (0, 0,
               "file %s has been modified, but has been removed in revision %s as of %s",
               finfo->fullname, jrev2, jdate2);
    else
2230    error (0, 0,
               "file %s has been modified, but has been removed in revision %s",
               finfo->fullname, jrev2);

    /* FIXME: Should we arrange to return a non-zero exit
       status? */

    if (rev1 != NULL)
        free (rev1);

2240    return;
}

if (rev1 != NULL)
    free (rev1);

/* The user file exists and has not been modified. Mark it
   for removal. FIXME: If we are doing a checkout, this has
   the effect of first checking out the file, and then
```

```
            removing it.  It would be better to just register the
            removal.  */
#ifdef SERVER_SUPPORT
        if (server_active)
        {
            server_scratch (finfo->file);
            server_updated (finfo, vers, SERVER_UPDATED, (mode_t) -1,
                            (unsigned char *) NULL, (struct buffer *) NULL);
        }
#endif
        mrev = xmalloc (strlen (vers->vn_user) + 2);
        sprintf (mrev, "-%s", vers->vn_user);
        Register (finfo->entries, finfo->file, mrev, vers->ts_rcs,
                  vers->options, vers->tag, vers->date, vers->ts_conflict, CVSroot_directory, finfo->repository);
        free (mrev);
        /* We need to check existence_error here because if we are
           running as the server, and the file is up to date in the
           working directory, the client will not have sent us a copy.  */
        if (unlink_file (finfo->file) < 0 && ! existence_error (errno))
            error (0, errno, "cannot remove file %s", finfo->fullname);
#ifdef SERVER_SUPPORT
        if (server_active)
            server_checked_in (finfo->file, finfo->update_dir,
                               finfo->repository);
#endif
        if (! really_quiet)
            error (0, 0, "scheduling %s for removal", finfo->fullname);

        return;
    }

    /* If the target of the merge is the same as the working file
       revision, then there is nothing to do.  */
    if (vers->vn_user != NULL && strcmp (rev2, vers->vn_user) == 0)
    {
        if (rev1 != NULL)
            free (rev1);
        free (rev2);
        return;
    }

    /* If rev1 is dead or does not exist, then the file was added
       between rev1 and rev2.  */
    if (rev1 == NULL || RCS_isdead (vers->srcfile, rev1))
    {
        if (rev1 != NULL)
            free (rev1);
        free (rev2);

        /* If the file does not exist in the working directory, then
           we can just check out the new revision and mark it for
           addition.  */
        if (vers->vn_user == NULL)
        {
            Vers_TS *xvers;

            xvers = Version_TS (finfo, vers->options, jrev2, jdate2, 1, 0);

            /* FIXME: If checkout_file fails, we should arrange to
               return a non-zero exit status.  */
            status = checkout_file (finfo, xvers, 1, 0, 1);

            freevers_ts (&xvers);

            return;
        }

        /* The file currently exists in the working directory, so we
           have a conflict which we can not resolve.  Note that this
           is true even if the file is marked for addition or removal.  */

        if (jdate2 != NULL)
            error (0, 0,
                   "file %s exists, but has been added in revision %s as of %s",
                   finfo->fullname, jrev2, jdate2);
        else
            error (0, 0,
                   "file %s exists, but has been added in revision %s",
                   finfo->fullname, jrev2);

        return;
    }

    /* If the two merge revisions are the same, then there is nothing
       to do.  */
    if (strcmp (rev1, rev2) == 0)
    {
        free (rev1);
        free (rev2);
        return;
```

```
    }

    /* If there is no working file, then we can't do the merge.  */
    if (vers->vn_user == NULL)
    {
        free (rev1);
        free (rev2);

        if (jdate2 != NULL)
            error (0, 0,
                   "file %s is present in revision %s as of %s",
                   finfo->fullname, jrev2, jdate2);
        else
            error (0, 0,
                   "file %s is present in revision %s",
                   finfo->fullname, jrev2);

        /* FIXME: Should we arrange to return a non-zero exit status?  */

        return;
    }

#ifdef SERVER_SUPPORT
    if (server_active && !isreadable (finfo->file))
    {
        int retcode;
        /* The file is up to date.  Need to check out the current contents.  */
        retcode = RCS_checkout (vers->srcfile, finfo->file,
                                vers->vn_user, (char *) NULL,
                                (char *) NULL, RUN_TTY,
                                (RCSCHECKOUTPROC) NULL, (void *) NULL);
        if (retcode != 0)
            error (1, 0,
                   "failed to check out %s file", finfo->fullname);
    }
#endif

    /*
     * The users currently modified file is moved to a backup file name
     * ".#filename.version", so that it will stay around for a few days
     * before being automatically removed by some cron daemon.  The "version"
     * is the version of the file that the user was most up-to-date with
     * before the merge.
     */
    backup = xmalloc (strlen (finfo->file)
                      + strlen (vers->vn_user)
                      + sizeof (BAKPREFIX)
                      + 10);
    (void) sprintf (backup, "%s%s.%s", BAKPREFIX, finfo->file, vers->vn_user);

    (void) unlink_file (backup);
    copy_file (finfo->file, backup);
    xchmod (finfo->file, 1);

    options = vers->options;
#if 0
    if (*options == '\0')
        options = "-kk";            /* to ignore keyword expansions */
#endif

    /* If the source of the merge is the same as the working file
       revision, then we can just RCS_checkout the target (no merging
       as such).  In the text file case, this is probably quite
       similar to the RCS_merge, but in the binary file case,
       RCS_merge gives all kinds of trouble.  */
    if (vers->vn_user != NULL
        && strcmp (rev1, vers->vn_user) == 0
        /* See comments above about how No_Difference has already been
           called.  */
        && vers->ts_user != NULL
        && strcmp (vers->ts_user, vers->ts_rcs) == 0

        /* This is because of the worry below about $Name.  If that
           isn't a problem, I suspect this code probably works for
           text files too.  */
        && (strcmp (options, "-kb") == 0
            || wrap_merge_is_copy (finfo->file)))
    {
        /* FIXME: what about nametag?  What does RCS_merge do with
           $Name?  */
        if (RCS_checkout (finfo->rcs, finfo->file, rev2, NULL, options,
                          RUN_TTY, (RCSCHECKOUTPROC)0, NULL) != 0)
            status = 2;
        else
            status = 0;

        /* OK, this is really stupid.  RCS_checkout carefully removes
           write permissions, and we carefully put them back.  But
           until someone gets around to fixing it, that seems like the
           easiest way to get what would seem to be the right mode.
```

```
            I don't check CVSWRITE or _watched; I haven't thought about
2430        that in great detail, but it seems like a watched file should
            be checked out (writable) after a merge.  */
        xchmod (finfo->file, 1);

        /* Traditionally, the text file case prints a whole bunch of
           scary looking and verbose output which fails to tell the user
           what is really going on (it gives them rev1 and rev2 but doesn't
           indicate in any way that rev1 == vn_user).  I think just a
           simple "U foo" is good here; it seems analogous to the case in
           which the file was added on the branch in terms of what to
2440       print.  */
        write_letter (finfo, 'U');
    }
    else if (strcmp (options, "-kb") == 0
             || wrap_merge_is_copy (finfo->file)
             || special_file_mismatch (finfo, rev1, rev2))
    {
        /* We are dealing with binary files, or files with a
           permission/linkage mismatch, and real merging would
           need to take place.  This is a conflict.  We give the user
2450       the two files, and let them resolve it.  It is possible
           that we should require a "touch foo" or similar step before
           we allow a checkin.  */
        if (RCS_checkout (finfo->rcs, finfo->file, rev2, NULL, options,
                          RUN_TTY, (RCSCHECKOUTPROC)0, NULL) != 0)
            status = 2;
        else
            status = 0;

        /* OK, this is really stupid.  RCS_checkout carefully removes
2460       write permissions, and we carefully put them back.  But
           until someone gets around to fixing it, that seems like the
           easiest way to get what would seem to be the right mode.
           I don't check CVSWRITE or _watched; I haven't thought about
           that in great detail, but it seems like a watched file should
           be checked out (writable) after a merge.  */
        xchmod (finfo->file, 1);

        /* Hmm.  We don't give them REV1 anywhere.  I guess most people
           probably don't have a 3-way merge tool for the file type in
2470       question, and might just get confused if we tried to either
           provide them with a copy of the file from REV1, or even just
           told them what REV1 is so they can get it themself, but it
           might be worth thinking about.  */
        /* See comment in merge_file about the "nonmergeable file"
           terminology.  */
        error (0, 0, "nonmergeable file needs merge");
        error (0, 0, "revision %s from repository is now in %s",
               rev2, finfo->fullname);
        error (0, 0, "file from working directory is now in %s", backup);
2480    write_letter (finfo, 'C');
    }
    else
        status = RCS_merge (finfo->rcs, vers->srcfile->path, finfo->file,
                            options, rev1, rev2);

    if (status != 0 && status != 1)
    {
        error (0, status == -1 ? errno : 0,
               "could not merge revision %s of %s", rev2, finfo->fullname);
2490    error (status == -1 ? 1 : 0, 0, "restoring %s from backup file %s",
               finfo->fullname, backup);
        rename_file (backup, finfo->file);
    }
    free (rev1);
    free (rev2);

    /* The file has changed, but if we just checked it out it may
       still have the same timestamp it did when it was first
       registered above in checkout_file.  We register it again with a
2500   dummy timestamp to make sure that later runs of CVS will
       recognize that it has changed.

       We don't actually need to register again if we called
       RCS_checkout above, and we aren't running as the server.
       However, that is not the normal case, and calling Register
       again won't cost much in that case.  */
    {
        char *cp = 0;

2510    if (status)
        {
            (void) time (&last_register_time);
            cp = time_stamp (finfo->file);
        }
        Register (finfo->entries, finfo->file, vers->vn_rcs,
                  "Result of merge", vers->options, vers->tag,
                  vers->date, cp, CVSroot_directory, finfo->repository);
        if (cp)
```

```
            free(cp);
    }

#ifdef SERVER_SUPPORT
    if (server_active)
    {
        server_copy_file (finfo->file, finfo->update_dir, finfo->repository,
                          backup);
        server_updated (finfo, vers, SERVER_MERGED,
                        (mode_t) -1, (unsigned char *) NULL,
                        (struct buffer *) NULL);
    }
#endif
    free (backup);
}

/*
 * Report whether revisions REV1 and REV2 of FINFO agree on:
 *   . file ownership
 *   . permissions
 *   . major and minor device numbers
 *   . symbolic links
 *   . hard links
 *
 * If either REV1 or REV2 is NULL, the working copy is used instead.
 *
 * Return 1 if the files differ on these data.
 */

int
special_file_mismatch (finfo, rev1, rev2)
    struct file_info *finfo;
    char *rev1;
    char *rev2;
{
#ifdef PRESERVE_PERMISSIONS_SUPPORT
    struct stat sb;
    RCSVers *vp;
    Node *n;
    uid_t rev1_uid, rev2_uid;
    gid_t rev1_gid, rev2_gid;
    mode_t rev1_mode, rev2_mode;
    unsigned long dev_long;
    dev_t rev1_dev, rev2_dev;
    char *rev1_symlink = NULL;
    char *rev2_symlink = NULL;
    List *rev1_hardlinks;
    List *rev2_hardlinks;
    int check_uids, check_gids, check_modes;
    int result;

    /* If we don't care about special file info, then
       don't report a mismatch in any case. */
    if (!preserve_perms)
        return 0;

    /* When special_file_mismatch is called from No_Difference, the
       RCS file has been only partially parsed.  We must read the
       delta tree in order to compare special file info recorded in
       the delta nodes.  (I think this is safe. -twp) */
    if (finfo->rcs->flags & PARTIAL)
        RCS_reparsercsfile (finfo->rcs, NULL, NULL);

    check_uids = check_gids = check_modes = 1;

    /* Obtain file information for REV1.  If this is null, then stat
       finfo->file and use that info. */
    /* If a revision does not know anything about its status,
       then presumably it doesn't matter, and indicates no conflict. */

    if (rev1 == NULL)
    {
        if (islink (finfo->file))
            rev1_symlink = xreadlink (finfo->file);
        else
        {
            if (CVS_LSTAT (finfo->file, &sb) < 0)
                error (1, errno, "could not get file information for %s",
                       finfo->file);
            rev1_uid = sb.st_uid;
            rev1_gid = sb.st_gid;
            rev1_mode = sb.st_mode;
            if (S_ISBLK (rev1_mode) || S_ISCHR (rev1_mode))
                rev1_dev = sb.st_rdev;
        }
        rev1_hardlinks = list_linked_files_on_disk (finfo->file);
    }
    else
    {
        n = findnode (finfo->rcs->versions, rev1);
```

```
        vp = (RCSVers *) n->data;

        n = findnode (vp->other_delta, "symlink");
        if (n != NULL)
            rev1_symlink = xstrdup (n->data);
        else
        {
            n = findnode (vp->other_delta, "owner");
            if (n == NULL)
                check_uids = 0; /* don't care */
            else
                rev1_uid = strtoul (n->data, NULL, 10);

            n = findnode (vp->other_delta, "group");
            if (n == NULL)
                check_gids = 0; /* don't care */
            else
                rev1_gid = strtoul (n->data, NULL, 10);

            n = findnode (vp->other_delta, "permissions");
            if (n == NULL)
                check_modes = 0;  /* don't care */
            else
                rev1_mode = strtoul (n->data, NULL, 8);

            n = findnode (vp->other_delta, "special");
            if (n == NULL)
                rev1_mode |= S_IFREG;
            else
            {
                /* If the size of 'ftype' changes, fix the sscanf call also */
                char ftype[16];
                if (sscanf (n->data, "%16s %lu", ftype,
                            &dev_long) < 2)
                    error (1, 0, "%s:%s has bad 'special' newphrase %s",
                           finfo->file, rev1, n->data);
                rev1_dev = dev_long;
                if (strcmp (ftype, "character") == 0)
                    rev1_mode |= S_IFCHR;
                else if (strcmp (ftype, "block") == 0)
                    rev1_mode |= S_IFBLK;
                else
                    error (0, 0, "%s:%s unknown file type '%s'",
                           finfo->file, rev1, ftype);
            }

            rev1_hardlinks = vp->hardlinks;
            if (rev1_hardlinks == NULL)
                rev1_hardlinks = getlist();
        }
    }

    /* Obtain file information for REV2. */
    if (rev2 == NULL)
    {
        if (islink (finfo->file))
            rev2_symlink = xreadlink (finfo->file);
        else
        {
            if (CVS_LSTAT (finfo->file, &sb) < 0)
                error (1, errno, "could not get file information for %s",
                       finfo->file);
            rev2_uid = sb.st_uid;
            rev2_gid = sb.st_gid;
            rev2_mode = sb.st_mode;
            if (S_ISBLK (rev2_mode) || S_ISCHR (rev2_mode))
                rev2_dev = sb.st_rdev;
        }
        rev2_hardlinks = list_linked_files_on_disk (finfo->file);
    }
    else
    {
        n = findnode (finfo->rcs->versions, rev2);
        vp = (RCSVers *) n->data;

        n = findnode (vp->other_delta, "symlink");
        if (n != NULL)
            rev2_symlink = xstrdup (n->data);
        else
        {
            n = findnode (vp->other_delta, "owner");
            if (n == NULL)
                check_uids = 0; /* don't care */
            else
                rev2_uid = strtoul (n->data, NULL, 10);

            n = findnode (vp->other_delta, "group");
            if (n == NULL)
                check_gids = 0; /* don't care */
            else
```

```
            rev2_gid = strtoul (n->data, NULL, 10);

        n = findnode (vp->other_delta, "permissions");
        if (n == NULL)
            check_modes = 0;  /* don't care */
        else
            rev2_mode = strtoul (n->data, NULL, 8);

        n = findnode (vp->other_delta, "special");
        if (n == NULL)
            rev2_mode |= S_IFREG;
        else
        {
            /* If the size of 'ftype' changes, fix the sscanf call also */
            char ftype[16];
            if (sscanf (n->data, "%16s %lu", ftype,
                        &dev_long) < 2)
                error (1, 0, "%s:%s has bad 'special' newphrase %s",
                       finfo->file, rev2, n->data);
            rev2_dev = dev_long;
            if (strcmp (ftype, "character") == 0)
                rev2_mode |= S_IFCHR;
            else if (strcmp (ftype, "block") == 0)
                rev2_mode |= S_IFBLK;
            else
                error (0, 0, "%s:%s unknown file type '%s'",
                       finfo->file, rev2, ftype);
        }

        rev2_hardlinks = vp->hardlinks;
        if (rev2_hardlinks == NULL)
            rev2_hardlinks = getlist();
    }
}

/* Check the user/group ownerships and file permissions, printing
   an error for each mismatch found.  Return 0 if all characteristics
   matched, and 1 otherwise. */

result = 0;

/* Compare symlinks first, since symlinks are simpler (don't have
   any other characteristics). */
if (rev1_symlink != NULL && rev2_symlink == NULL)
{
    error (0, 0, "%s is a symbolic link",
           (rev1 == NULL ? "working file" : rev1));
    result = 1;
}
else if (rev1_symlink == NULL && rev2_symlink != NULL)
{
    error (0, 0, "%s is a symbolic link",
           (rev2 == NULL ? "working file" : rev2));
    result = 1;
}
else if (rev1_symlink != NULL)
    result = (strcmp (rev1_symlink, rev2_symlink) == 0);
else
{
    /* Compare user ownership. */
    if (check_uids && rev1_uid != rev2_uid)
    {
        error (0, 0, "%s: owner mismatch between %s and %s",
               finfo->file,
               (rev1 == NULL ? "working file" : rev1),
               (rev2 == NULL ? "working file" : rev2));
        result = 1;
    }

    /* Compare group ownership. */
    if (check_gids && rev1_gid != rev2_gid)
    {
        error (0, 0, "%s: group mismatch between %s and %s",
               finfo->file,
               (rev1 == NULL ? "working file" : rev1),
               (rev2 == NULL ? "working file" : rev2));
        result = 1;
    }

    /* Compare permissions. */
    if (check_modes &&
        (rev1_mode & 07777) != (rev2_mode & 07777))
    {
        error (0, 0, "%s: permission mismatch between %s and %s",
               finfo->file,
               (rev1 == NULL ? "working file" : rev1),
               (rev2 == NULL ? "working file" : rev2));
        result = 1;
    }
```

```
        /* Compare device file characteristics. */
2790    if ((rev1_mode & S_IFMT) != (rev2_mode & S_IFMT))
        {
            error (0, 0, "%s: %s and %s are different file types",
                   finfo->file,
                   (rev1 == NULL ? "working file" : rev1),
                   (rev2 == NULL ? "working file" : rev2));
            result = 1;
        }
        else if (S_ISBLK (rev1_mode))
        {
2800        if (rev1_dev != rev2_dev)
            {
                error (0, 0, "%s: device numbers of %s and %s do not match",
                       finfo->file,
                       (rev1 == NULL ? "working file" : rev1),
                       (rev2 == NULL ? "working file" : rev2));
                result = 1;
            }
        }

2810    /* Compare hard links. */
        if (compare_linkage_lists (rev1_hardlinks, rev2_hardlinks) == 0)
        {
            error (0, 0, "%s: hard linkage of %s and %s do not match",
                   finfo->file,
                   (rev1 == NULL ? "working file" : rev1),
                   (rev2 == NULL ? "working file" : rev2));
            result = 1;
        }
    }
2820
    if (rev1_symlink != NULL)
        free (rev1_symlink);
    if (rev2_symlink != NULL)
        free (rev2_symlink);
    if (rev1_hardlinks != NULL)
        dellist (&rev1_hardlinks);
    if (rev2_hardlinks != NULL)
        dellist (&rev2_hardlinks);

2830    return result;
#else
    return 0;
#endif
}

int
joining ()
{
    return (join_rev1 != NULL);
2840 }
```

A.61 update.h

```
/* Declarations for update.c.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

int do_update PROTO((int argc, char *argv[], char *xoptions, char *xtag,
                     char *xdate, int xforce, int local, int xbuild,
                     int xaflag, int xprune, int xpipeout, int which,
                     char *xjoin_rev1, char *xjoin_rev2, char *preload_update_dir));
int joining PROTO((void));
extern int isemptydir PROTO ((char *dir, int might_not_exist));
```

A.62 vers_ts.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"

#ifdef SERVER_SUPPORT
static void time_stamp_server PROTO((char *, Vers_TS *, Entnode *));
#endif

/* Fill in and return a Vers_TS structure for the file FINFO.  TAG and
   DATE are from the command line.  */

Vers_TS *
Version_TS (finfo, options, tag, date, force_tag_match, set_time)
    struct file_info *finfo;

    /* Keyword expansion options, I think generally from the command
       line.  Can be either NULL or "" to indicate none are specified
       here.  */
    char *options;
    char *tag;
    char *date;
    int force_tag_match;
    int set_time;
{
    Node *p;
    RCSNode *rcsdata;
    Vers_TS *vers_ts;
    struct stickydirtag *sdtp;
    Entnode *entdata;

    /* get a new Vers_TS struct */
    vers_ts = (Vers_TS *) xmalloc (sizeof (Vers_TS));
    memset ((char *) vers_ts, 0, sizeof (*vers_ts));

    /*
     * look up the entries file entry and fill in the version and timestamp
     * if entries is NULL, there is no entries file so don't bother trying to
     * look it up (used by checkout -P)
     */
    if (finfo->entries == NULL)
    {
        sdtp = NULL;
        p = NULL;
    }
    else
    {
        p = findnode_fn (finfo->entries, finfo->file);
        sdtp = (struct stickydirtag *) finfo->entries->list->data; /* list-private */
    }

    entdata = NULL;
    if (p != NULL)
    {
        entdata = (Entnode *) p->data;

        if (entdata->type == ENT_SUBDIR)
        {
            /* According to cvs.texinfo, the various fields in the Entries
               file for a directory (other than the name) do not have a
               defined meaning.  We need to pass them along without getting
               confused based on what is in them.  Therefore we make sure
               not to set vn_user and the like from Entries, add.c and
               perhaps other code will expect these fields to be NULL for
               a directory.  */
            vers_ts->entdata = entdata;
        }
        else
#ifdef SERVER_SUPPORT
        /* An entries line with "D" in the timestamp indicates that the
           client sent Is-modified without sending Entry.  So we want to
           use the entries line for the sole purpose of telling
           time_stamp_server what is up; we don't want the rest of CVS
           to think there is an entries line.  */
        if (strcmp (entdata->timestamp, "D") != 0)
#endif
        {
            vers_ts->vn_user = xstrdup (entdata->version);
            vers_ts->ts_rcs = xstrdup (entdata->timestamp);
            vers_ts->ts_conflict = xstrdup (entdata->conflict);
            if (!tag)
            {
                if (!(sdtp && sdtp->aflag))
```

```
                vers_ts->tag = xstrdup (entdata->tag);
90          }
            if (!date)
            {
                if (!(sdtp && sdtp->aflag))
                    vers_ts->date = xstrdup (entdata->date);
            }
            vers_ts->entdata = entdata;
        }
        /* Even if we don't have an "entries line" as such
           (vers_ts->entdata), we want to pick up options which could
100        have been from a Kopt protocol request.  */
        if (!options || (options && *options == '\0'))
        {
            if (!(sdtp && sdtp->aflag))
                vers_ts->options = xstrdup (entdata->options);
        }
    }

    /*
     * -k options specified on the command line override (and overwrite)
110  * options stored in the entries file
     */
    if (options && *options != '\0')
        vers_ts->options = xstrdup (options);
    else if (!vers_ts->options || *vers_ts->options == '\0')
    {
        if (finfo->rcs != NULL)
        {
            /* If no keyword expansion was specified on command line,
               use whatever was in the rcs file (if there is one).  This
120            is how we, if we are the server, tell the client whether
               a file is binary.  */
            char *rcsexpand = RCS_getexpand (finfo->rcs);
            if (rcsexpand != NULL)
            {
                vers_ts->options = xmalloc (strlen (rcsexpand) + 3);
                strcpy (vers_ts->options, "-k");
                strcat (vers_ts->options, rcsexpand);
            }
        }
130 }
    if (!vers_ts->options)
        vers_ts->options = xstrdup ("");

    /*
     * if tags were specified on the command line, they override what is in
     * the Entries file
     */
    if (tag || date)
    {
140     vers_ts->tag = xstrdup (tag);
        vers_ts->date = xstrdup (date);
    }
    else if (!vers_ts->entdata && (sdtp && sdtp->aflag == 0))
    {
        if (!vers_ts->tag)
        {
            vers_ts->tag = xstrdup (sdtp->tag);
            vers_ts->nonbranch = sdtp->nonbranch;
        }
150     if (!vers_ts->date)
            vers_ts->date = xstrdup (sdtp->date);
    }

    /* Now look up the info on the source controlled file */
    if (finfo->rcs != NULL)
    {
        rcsdata = finfo->rcs;
        rcsdata->refcount++;
    }
160 else if (finfo->repository != NULL)
        rcsdata = RCS_parse (finfo->file, finfo->repository);
    else
        rcsdata = NULL;

    if (rcsdata != NULL)
    {
        /* squirrel away the rcsdata pointer for others */
        vers_ts->srcfile = rcsdata;

170     if (vers_ts->tag && strcmp (vers_ts->tag, TAG_BASE) == 0)
        {
            vers_ts->vn_rcs = xstrdup (vers_ts->vn_user);
            vers_ts->vn_tag = xstrdup (vers_ts->vn_user);
        }
        else
        {
            int simple;
            char* local_rev = NULL;
```

```
            /* We try remote first, because if the version is remote, and branched
               at a local revision, we want to return remote, not local */
            vers_ts->vn_rcs = RCS_getremoteversion (finfo, rcsdata, vers_ts->tag, &local_rev);
            if (vers_ts->vn_rcs == NULL) {
              /* Not remote, try regular (local) */
              if (local_rev != NULL) {
                vers_ts->tag = local_rev;
              }
            vers_ts->vn_rcs = RCS_getversion (rcsdata, vers_ts->tag,
                                              vers_ts->date, force_tag_match,
                                              &simple);
              if (vers_ts->vn_rcs == NULL) {
                vers_ts->vn_tag = NULL;
              }
            else if (simple)
                vers_ts->vn_tag = xstrdup (vers_ts->tag);
            else
                vers_ts->vn_tag = xstrdup (vers_ts->vn_rcs);
            } else {
              /* Found remote, forget about local */
                if (vers_ts->vn_rcs != NULL) {
                  vers_ts->vn_remote = xstrdup (vers_ts->vn_rcs);
                }
            }
        }

        /*
         * If the source control file exists and has the requested revision,
         * get the Date the revision was checked in.  If "user" exists, set
         * its mtime.
         */
        if (set_time && vers_ts->vn_rcs != NULL)
        {
#ifdef SERVER_SUPPORT
            if (server_active)
                server_modtime (finfo, vers_ts);
            else
#endif
            {
                struct utimbuf t;

                memset (&t, 0, sizeof (t));
                t.modtime =
                    RCS_getrevtime (rcsdata, vers_ts->vn_rcs, 0, 0);
                if (t.modtime != (time_t) -1)
                {
                    t.actime = t.modtime;

                    /* This used to need to ignore existence_errors
                       (for cases like where update.c now clears
                       set_time if noexec, but didn't used to).  I
                       think maybe now it doesn't (server_modtime does
                       not like those kinds of cases).  */
                    (void) utime (finfo->file, &t);
                }
            }
        }
    }

    /* get user file time-stamp in ts_user */
    if (finfo->entries != (List *) NULL)
    {
#ifdef SERVER_SUPPORT
        if (server_active)
            time_stamp_server (finfo->file, vers_ts, entdata);
        else
#endif
            vers_ts->ts_user = time_stamp (finfo->file);
    }

    return (vers_ts);
}

#ifdef SERVER_SUPPORT

/* Set VERS_TS->TS_USER to time stamp for FILE.  */

/* Separate these out to keep the logic below clearer.  */
#define mark_lost(V)      ((V)->ts_user = 0)
#define mark_unchanged(V) ((V)->ts_user = xstrdup ((V)->ts_rcs))

static void
time_stamp_server (file, vers_ts, entdata)
    char *file;
    Vers_TS *vers_ts;
    Entnode *entdata;
{
    struct stat sb;
    char *cp;
```

```
    if (CVS_LSTAT (file, &sb) < 0)
    {
        if (! existence_error (errno))
            error (1, errno, "cannot stat temp file");

        /* Missing file means lost or unmodified; check entries
           file to see which.

           XXX FIXME - If there's no entries file line, we
           wouldn't be getting the file at all, so consider it
           lost.  I don't know that that's right, but it's not
           clear to me that either choice is.  Besides, would we
           have an RCS string in that case anyways?  */
        if (entdata == NULL)
            mark_lost (vers_ts);
        else if (entdata->timestamp
                 && entdata->timestamp[0] == '=')
            mark_unchanged (vers_ts);
        else if (entdata->timestamp != NULL
                 && (entdata->timestamp[0] == 'M'
                     || entdata->timestamp[0] == 'D')
                 && entdata->timestamp[1] == '\0')
            vers_ts->ts_user = xstrdup ("Is-modified");
        else
            mark_lost (vers_ts);
    }
    else if (sb.st_mtime == 0)
    {
        /* We shouldn't reach this case any more!  */
        abort ();
    }
    else
    {
        struct tm *tm_p;
        struct tm local_tm;

        vers_ts->ts_user = xmalloc (25);
        /* We want to use the same timestamp format as is stored in the
           st_mtime.  For unix (and NT I think) this *must* be universal
           time (UT), so that files don't appear to be modified merely
           because the timezone has changed.  For VMS, or hopefully other
           systems where gmtime returns NULL, the modification time is
           stored in local time, and therefore it is not possible to cause
           st_mtime to be out of sync by changing the timezone.  */
        tm_p = gmtime (&sb.st_mtime);
        if (tm_p)
        {
            memcpy (&local_tm, tm_p, sizeof (local_tm));
            cp = asctime (&local_tm); /* copy in the modify time */
        }
        else
            cp = ctime (&sb.st_mtime);

        cp[24] = 0;
        (void) strcpy (vers_ts->ts_user, cp);
    }
}

#endif /* SERVER_SUPPORT */
/*
 * Gets the time-stamp for the file "file" and returns it in space it
 * allocates
 */
char *
time_stamp (file)
    char *file;
{
    struct stat sb;
    char *cp;
    char *ts;

    if (CVS_LSTAT (file, &sb) < 0)
    {
        ts = NULL;
    }
    else
    {
        struct tm *tm_p;
        struct tm local_tm;
        ts = xmalloc (25);
        /* We want to use the same timestamp format as is stored in the
           st_mtime.  For unix (and NT I think) this *must* be universal
           time (UT), so that files don't appear to be modified merely
           because the timezone has changed.  For VMS, or hopefully other
           systems where gmtime returns NULL, the modification time is
           stored in local time, and therefore it is not possible to cause
           st_mtime to be out of sync by changing the timezone.  */
        tm_p = gmtime (&sb.st_mtime);
        if (tm_p)
```

```
        {
360         memcpy (&local_tm, tm_p, sizeof (local_tm));
            cp = asctime (&local_tm); /* copy in the modify time */
        }
        else
            cp = ctime(&sb.st_mtime);

        cp[24] = 0;
        (void) strcpy (ts, cp);
    }

370 return (ts);
}

/*
 * free up a Vers_TS struct
 */
void
freevers_ts (versp)
    Vers_TS **versp;
{
380 if ((*versp)->srcfile)
        freercsnode (&((*versp)->srcfile));
    if ((*versp)->vn_user)
        free ((*versp)->vn_user);
    if ((*versp)->vn_rcs)
        free ((*versp)->vn_rcs);
    if ((*versp)->vn_tag)
        free ((*versp)->vn_tag);
    if ((*versp)->ts_user)
        free ((*versp)->ts_user);
390 if ((*versp)->ts_rcs)
        free ((*versp)->ts_rcs);
    if ((*versp)->options)
        free ((*versp)->options);
    if ((*versp)->tag)
        free ((*versp)->tag);
    if ((*versp)->date)
        free ((*versp)->date);
    if ((*versp)->ts_conflict)
        free ((*versp)->ts_conflict);
400 free ((char *) *versp);
    *versp = (Vers_TS *) NULL;
}
```

A.63 vers_ts.c

```
/*
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with the CVS source distribution.
 */

#include "cvs.h"

#ifdef SERVER_SUPPORT
static void time_stamp_server PROTO((char *, Vers_TS *, Entnode *));
#endif

/* Fill in and return a Vers_TS structure for the file FINFO.  TAG and
   DATE are from the command line.  */

Vers_TS *
Version_TS (finfo, options, tag, date, force_tag_match, set_time)
    struct file_info *finfo;

    /* Keyword expansion options, I think generally from the command
       line.  Can be either NULL or "" to indicate none are specified
       here.  */
    char *options;
    char *tag;
    char *date;
    int force_tag_match;
    int set_time;
{
    Node *p;
    RCSNode *rcsdata;
    Vers_TS *vers_ts;
    struct stickydirtag *sdtp;
    Entnode *entdata;

    /* get a new Vers_TS struct */
    vers_ts = (Vers_TS *) xmalloc (sizeof (Vers_TS));
    memset ((char *) vers_ts, 0, sizeof (*vers_ts));

    /*
     * look up the entries file entry and fill in the version and timestamp
     * if entries is NULL, there is no entries file so don't bother trying to
     * look it up (used by checkout -P)
     */
    if (finfo->entries == NULL)
    {
        sdtp = NULL;
        p = NULL;
    }
    else
    {
        p = findnode_fn (finfo->entries, finfo->file);
        sdtp = (struct stickydirtag *) finfo->entries->list->data; /* list-private */
    }

    entdata = NULL;
    if (p != NULL)
    {
        entdata = (Entnode *) p->data;

        if (entdata->type == ENT_SUBDIR)
        {
            /* According to cvs.texinfo, the various fields in the Entries
               file for a directory (other than the name) do not have a
               defined meaning.  We need to pass them along without getting
               confused based on what is in them.  Therefore we make sure
               not to set vn_user and the like from Entries, add.c and
               perhaps other code will expect these fields to be NULL for
               a directory.  */
            vers_ts->entdata = entdata;
        }
        else
#ifdef SERVER_SUPPORT
        /* An entries line with "D" in the timestamp indicates that the
           client sent Is-modified without sending Entry.  So we want to
           use the entries line for the sole purpose of telling
           time_stamp_server what is up; we don't want the rest of CVS
           to think there is an entries line.  */
        if (strcmp (entdata->timestamp, "D") != 0)
#endif
        {
            vers_ts->vn_user = xstrdup (entdata->version);
            vers_ts->ts_rcs = xstrdup (entdata->timestamp);
            vers_ts->ts_conflict = xstrdup (entdata->conflict);
            if (!tag)
            {
                if (!(sdtp && sdtp->aflag))
```

```
                vers_ts->tag = xstrdup (entdata->tag);
        }
        if (!date)
        {
            if (!(sdtp && sdtp->aflag))
                vers_ts->date = xstrdup (entdata->date);
        }
        vers_ts->entdata = entdata;
    }
    /* Even if we don't have an "entries line" as such
       (vers_ts->entdata), we want to pick up options which could
       have been from a Kopt protocol request.  */
    if (!options || (options && *options == '\0'))
    {
        if (!(sdtp && sdtp->aflag))
            vers_ts->options = xstrdup (entdata->options);
    }
}

/*
 * -k options specified on the command line override (and overwrite)
 * options stored in the entries file
 */
if (options && *options != '\0')
    vers_ts->options = xstrdup (options);
else if (!vers_ts->options || *vers_ts->options == '\0')
{
    if (finfo->rcs != NULL)
    {
        /* If no keyword expansion was specified on command line,
           use whatever was in the rcs file (if there is one).  This
           is how we, if we are the server, tell the client whether
           a file is binary.  */
        char *rcsexpand = RCS_getexpand (finfo->rcs);
        if (rcsexpand != NULL)
        {
            vers_ts->options = xmalloc (strlen (rcsexpand) + 3);
            strcpy (vers_ts->options, "-k");
            strcat (vers_ts->options, rcsexpand);
        }
    }
}
if (!vers_ts->options)
    vers_ts->options = xstrdup ("");

/*
 * if tags were specified on the command line, they override what is in
 * the Entries file
 */
if (tag || date)
{
    vers_ts->tag = xstrdup (tag);
    vers_ts->date = xstrdup (date);
}
else if (!vers_ts->entdata && (sdtp && sdtp->aflag == 0))
{
    if (!vers_ts->tag)
    {
        vers_ts->tag = xstrdup (sdtp->tag);
        vers_ts->nonbranch = sdtp->nonbranch;
    }
    if (!vers_ts->date)
        vers_ts->date = xstrdup (sdtp->date);
}

/* Now look up the info on the source controlled file */
if (finfo->rcs != NULL)
{
    rcsdata = finfo->rcs;
    rcsdata->refcount++;
}
else if (finfo->repository != NULL)
    rcsdata = RCS_parse (finfo->file, finfo->repository);
else
    rcsdata = NULL;

if (rcsdata != NULL)
{
    /* squirrel away the rcsdata pointer for others */
    vers_ts->srcfile = rcsdata;

    if (vers_ts->tag && strcmp (vers_ts->tag, TAG_BASE) == 0)
    {
        vers_ts->vn_rcs = xstrdup (vers_ts->vn_user);
        vers_ts->vn_tag = xstrdup (vers_ts->vn_user);
    }
    else
    {
        int simple;
        char* local_rev = NULL;
```

```
            /* We try remote first, because if the version is remote, and branched
               at a local revision, we want to return remote, not local */
            vers_ts->vn_rcs = RCS_getremoteversion (finfo, rcsdata, vers_ts->tag, &local_rev);
            if (vers_ts->vn_rcs == NULL) {
              /* Not remote, try regular (local) */
              if (local_rev != NULL) {
                vers_ts->tag = local_rev;
              }
            vers_ts->vn_rcs = RCS_getversion (rcsdata, vers_ts->tag,
                                              vers_ts->date, force_tag_match,
                                              &simple);
              if (vers_ts->vn_rcs == NULL) {
                vers_ts->vn_tag = NULL;
              }
            else if (simple)
                vers_ts->vn_tag = xstrdup (vers_ts->tag);
            else
                vers_ts->vn_tag = xstrdup (vers_ts->vn_rcs);
            } else {
              /* Found remote, forget about local */
                if (vers_ts->vn_rcs != NULL) {
                  vers_ts->vn_remote = xstrdup (vers_ts->vn_rcs);
                }
            }
        }

        /*
         * If the source control file exists and has the requested revision,
         * get the Date the revision was checked in.  If "user" exists, set
         * its mtime.
         */
        if (set_time && vers_ts->vn_rcs != NULL)
        {
#ifdef SERVER_SUPPORT
            if (server_active)
                server_modtime (finfo, vers_ts);
            else
#endif
            {
                struct utimbuf t;

                memset (&t, 0, sizeof (t));
                t.modtime =
                    RCS_getrevtime (rcsdata, vers_ts->vn_rcs, 0, 0);
                if (t.modtime != (time_t) -1)
                {
                    t.actime = t.modtime;

                    /* This used to need to ignore existence_errors
                       (for cases like where update.c now clears
                       set_time if noexec, but didn't used to).  I
                       think maybe now it doesn't (server_modtime does
                       not like those kinds of cases).  */
                    (void) utime (finfo->file, &t);
                }
            }
        }
    }

    /* get user file time-stamp in ts_user */
    if (finfo->entries != (List *) NULL)
    {
#ifdef SERVER_SUPPORT
        if (server_active)
            time_stamp_server (finfo->file, vers_ts, entdata);
        else
#endif
            vers_ts->ts_user = time_stamp (finfo->file);
    }

    return (vers_ts);
}

#ifdef SERVER_SUPPORT

/* Set VERS_TS->TS_USER to time stamp for FILE.  */

/* Separate these out to keep the logic below clearer.  */
#define mark_lost(V)      ((V)->ts_user = 0)
#define mark_unchanged(V) ((V)->ts_user = xstrdup ((V)->ts_rcs))

static void
time_stamp_server (file, vers_ts, entdata)
    char *file;
    Vers_TS *vers_ts;
    Entnode *entdata;
{
    struct stat sb;
    char *cp;
```

```
    if (CVS_LSTAT (file, &sb) < 0)
    {
        if (! existence_error (errno))
            error (1, errno, "cannot stat temp file");

        /* Missing file means lost or unmodified; check entries
           file to see which.

           XXX FIXME - If there's no entries file line, we
           wouldn't be getting the file at all, so consider it
           lost.  I don't know that that's right, but it's not
           clear to me that either choice is.  Besides, would we
           have an RCS string in that case anyways?  */
        if (entdata == NULL)
            mark_lost (vers_ts);
        else if (entdata->timestamp
                 && entdata->timestamp[0] == '=')
            mark_unchanged (vers_ts);
        else if (entdata->timestamp != NULL
                 && (entdata->timestamp[0] == 'M'
                     || entdata->timestamp[0] == 'D')
                 && entdata->timestamp[1] == '\0')
            vers_ts->ts_user = xstrdup ("Is-modified");
        else
            mark_lost (vers_ts);
    }
    else if (sb.st_mtime == 0)
    {
        /* We shouldn't reach this case any more!  */
        abort ();
    }
    else
    {
        struct tm *tm_p;
        struct tm local_tm;

        vers_ts->ts_user = xmalloc (25);
        /* We want to use the same timestamp format as is stored in the
           st_mtime.  For unix (and NT I think) this *must* be universal
           time (UT), so that files don't appear to be modified merely
           because the timezone has changed.  For VMS, or hopefully other
           systems where gmtime returns NULL, the modification time is
           stored in local time, and therefore it is not possible to cause
           st_mtime to be out of sync by changing the timezone.  */
        tm_p = gmtime (&sb.st_mtime);
        if (tm_p)
        {
            memcpy (&local_tm, tm_p, sizeof (local_tm));
            cp = asctime (&local_tm); /* copy in the modify time */
        }
        else
            cp = ctime (&sb.st_mtime);

        cp[24] = 0;
        (void) strcpy (vers_ts->ts_user, cp);
    }
}

#endif /* SERVER_SUPPORT */
/*
 * Gets the time-stamp for the file "file" and returns it in space it
 * allocates
 */
char *
time_stamp (file)
    char *file;
{
    struct stat sb;
    char *cp;
    char *ts;

    if (CVS_LSTAT (file, &sb) < 0)
    {
        ts = NULL;
    }
    else
    {
        struct tm *tm_p;
        struct tm local_tm;
        ts = xmalloc (25);
        /* We want to use the same timestamp format as is stored in the
           st_mtime.  For unix (and NT I think) this *must* be universal
           time (UT), so that files don't appear to be modified merely
           because the timezone has changed.  For VMS, or hopefully other
           systems where gmtime returns NULL, the modification time is
           stored in local time, and therefore it is not possible to cause
           st_mtime to be out of sync by changing the timezone.  */
        tm_p = gmtime (&sb.st_mtime);
        if (tm_p)
```

```
        {
360         memcpy (&local_tm, tm_p, sizeof (local_tm));
            cp = asctime (&local_tm); /* copy in the modify time */
        }
        else
            cp = ctime(&sb.st_mtime);

        cp[24] = 0;
        (void) strcpy (ts, cp);
    }

370 return (ts);
}

/*
 * free up a Vers_TS struct
 */
void
freevers_ts (versp)
    Vers_TS **versp;
{
380 if ((*versp)->srcfile)
        freercsnode (&((*versp)->srcfile));
    if ((*versp)->vn_user)
        free ((*versp)->vn_user);
    if ((*versp)->vn_rcs)
        free ((*versp)->vn_rcs);
    if ((*versp)->vn_tag)
        free ((*versp)->vn_tag);
    if ((*versp)->ts_user)
        free ((*versp)->ts_user);
390 if ((*versp)->ts_rcs)
        free ((*versp)->ts_rcs);
    if ((*versp)->options)
        free ((*versp)->options);
    if ((*versp)->tag)
        free ((*versp)->tag);
    if ((*versp)->date)
        free ((*versp)->date);
    if ((*versp)->ts_conflict)
        free ((*versp)->ts_conflict);
400 free ((char *) *versp);
    *versp = (Vers_TS *) NULL;
}
```

A.64 version.c

```
/*
 * Copyright (c) 1994 david d 'zoo' zuhn
 * Copyright (c) 1994 Free Software Foundation, Inc.
 * Copyright (c) 1992, Brian Berliner and Jeff Polk
 * Copyright (c) 1989-1992, Brian Berliner
 *
 * You may distribute under the terms of the GNU General Public License as
 * specified in the README file that comes with this  CVS source distribution.
 *
 * version.c - the CVS version number
 */

#include "cvs.h"

/* NOTE: remember to remove 'Halibut' when patching this code. */
char *version_string = "\nConcurrent Versions System (CVS) 1.10 'Halibut'";

#ifdef CLIENT_SUPPORT
#ifdef SERVER_SUPPORT
char *config_string = " (client/server)\n";
#else
char *config_string = " (client)\n";
#endif
#else
#ifdef SERVER_SUPPORT
char *config_string = " (server)\n";
#else
char *config_string = "\n";
#endif
#endif
```

A.65 watch.c

```
/* Implementation for "cvs watch add", "cvs watchers", and related commands

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include "cvs.h"
#include "edit.h"
#include "fileattr.h"
#include "watch.h"

const char *const watch_usage[] =
{
    "Usage: %s %s [on|off|add|remove] [-lR] [-a action] [files...]\n",
    "on/off: turn on/off read-only checkouts of files\n",
    "add/remove: add or remove notification on actions\n",
    "-l (on/off/add/remove): Local directory only, not recursive\n",
    "-R (on/off/add/remove): Process directories recursively\n",
    "-a (add/remove): Specify what actions, one of\n",
    "   edit,unedit,commit,all,none\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

static struct addremove_args the_args;

void
watch_modify_watchers (file, what)
    char *file;
    struct addremove_args *what;
{
    char *curattr = fileattr_get0 (file, "_watchers");
    char *p;
    char *pend;
    char *nextp;
    char *who;
    int who_len;
    char *mycurattr;
    char *mynewattr;
    size_t mynewattr_size;

    int add_edit_pending;
    int add_unedit_pending;
    int add_commit_pending;
    int remove_edit_pending;
    int remove_unedit_pending;
    int remove_commit_pending;
    int add_tedit_pending;
    int add_tunedit_pending;
    int add_tcommit_pending;

    who = getcaller ();
    who_len = strlen (who);

    /* Look for current watcher types for this user.  */
    mycurattr = NULL;
    if (curattr != NULL)
    {
        p = curattr;
        while (1) {
            if (strncmp (who, p, who_len) == 0
                && p[who_len] == '>')
            {
                /* Found this user.  */
                mycurattr = p + who_len + 1;
            }
            p = strchr (p, ',');
            if (p == NULL)
                break;
            ++p;
        }
    }
    if (mycurattr != NULL)
    {
        mycurattr = xstrdup (mycurattr);
        p = strchr (mycurattr, ',');
        if (p != NULL)
            *p = '\0';
    }

    /* Now copy mycurattr to mynewattr, making the requisite modifications.
       Note that we add a dummy '+' to the start of mynewattr, to reduce
```

```
    special cases (but then we strip it off when we are done).  */
90
  mynewattr_size = sizeof "+edit+unedit+commit+tedit+tunedit+tcommit";
  if (mycurattr != NULL)
    mynewattr_size += strlen (mycurattr);
  mynewattr = xmalloc (mynewattr_size);
  mynewattr[0] = '\0';

  add_edit_pending = what->adding && what->edit;
  add_unedit_pending = what->adding && what->unedit;
  add_commit_pending = what->adding && what->commit;
100 remove_edit_pending = !what->adding && what->edit;
  remove_unedit_pending = !what->adding && what->unedit;
  remove_commit_pending = !what->adding && what->commit;
  add_tedit_pending = what->add_tedit;
  add_tunedit_pending = what->add_tunedit;
  add_tcommit_pending = what->add_tcommit;

  /* Copy over existing watch types, except those to be removed.  */
  p = mycurattr;
  while (p != NULL)
110 {
    pend = strchr (p, '+');
    if (pend == NULL)
    {
      pend = p + strlen (p);
      nextp = NULL;
    }
    else
      nextp = pend + 1;

120 /* Process this item.  */
    if (pend - p == 4 && strncmp ("edit", p, 4) == 0)
    {
      if (!remove_edit_pending)
        strcat (mynewattr, "+edit");
      add_edit_pending = 0;
    }
    else if (pend - p == 6 && strncmp ("unedit", p, 6) == 0)
    {
      if (!remove_unedit_pending)
130     strcat (mynewattr, "+unedit");
      add_unedit_pending = 0;
    }
    else if (pend - p == 6 && strncmp ("commit", p, 6) == 0)
    {
      if (!remove_commit_pending)
        strcat (mynewattr, "+commit");
      add_commit_pending = 0;
    }
    else if (pend - p == 5 && strncmp ("tedit", p, 5) == 0)
140 {
      if (!what->remove_temp)
        strcat (mynewattr, "+tedit");
      add_tedit_pending = 0;
    }
    else if (pend - p == 7 && strncmp ("tunedit", p, 7) == 0)
    {
      if (!what->remove_temp)
        strcat (mynewattr, "+tunedit");
      add_tunedit_pending = 0;
150 }
    else if (pend - p == 7 && strncmp ("tcommit", p, 7) == 0)
    {
      if (!what->remove_temp)
        strcat (mynewattr, "+tcommit");
      add_tcommit_pending = 0;
    }
    else
    {
      char *mp;
160
      /* Copy over any unrecognized watch types, for future
         expansion.  */
      mp = mynewattr + strlen (mynewattr);
      *mp++ = '+';
      strncpy (mp, p, pend - p);
      *(mp + (pend - p)) = '\0';
    }

    /* Set up for next item.  */
170 p = nextp;
  }

  /* Add in new watch types.  */
  if (add_edit_pending)
    strcat (mynewattr, "+edit");
  if (add_unedit_pending)
    strcat (mynewattr, "+unedit");
  if (add_commit_pending)
```

```
        strcat (mynewattr, "+commit");
    if (add_tedit_pending)
        strcat (mynewattr, "+tedit");
    if (add_tunedit_pending)
        strcat (mynewattr, "+tunedit");
    if (add_tcommit_pending)
        strcat (mynewattr, "+tcommit");

    {
        char *curattr_new;

        curattr_new =
          fileattr_modify (curattr,
                           who,
                           mynewattr[0] == '\0' ? NULL : mynewattr + 1,
                           '>',
                           ',');
        /* If the attribute is unchanged, don't rewrite the attribute file.  */
        if (!((curattr_new == NULL && curattr == NULL)
              || (curattr_new != NULL
                  && curattr != NULL
                  && strcmp (curattr_new, curattr) == 0)))
          fileattr_set (file,
                        "_watchers",
                        curattr_new);
        if (curattr_new != NULL)
            free (curattr_new);
    }

    if (curattr != NULL)
        free (curattr);
    if (mycurattr != NULL)
        free (mycurattr);
    if (mynewattr != NULL)
        free (mynewattr);
}

static int addremove_fileproc PROTO ((void *callerdat,
                                       struct file_info *finfo));

static int
addremove_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    watch_modify_watchers (finfo->file, &the_args);
    return 0;
}

static int addremove_filesdoneproc PROTO ((void *, int, char *, char *,
                                            List *));

static int
addremove_filesdoneproc (callerdat, err, repository, update_dir, entries)
    void *callerdat;
    int err;
    char *repository;
    char *update_dir;
    List *entries;
{
    if (the_args.setting_default)
        watch_modify_watchers (NULL, &the_args);
    return err;
}

static int watch_addremove PROTO ((int argc, char **argv));

static int
watch_addremove (argc, argv)
    int argc;
    char **argv;
{
    int c;
    int local = 0;
    int err;
    int a_omitted;

    a_omitted = 1;
    the_args.commit = 0;
    the_args.edit = 0;
    the_args.unedit = 0;
    optind = 0;
    while ((c = getopt (argc, argv, "+lRa:")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
```

```
                local = 0;
                break;
            case 'a':
                a_omitted = 0;
                if (strcmp (optarg, "edit") == 0)
                    the_args.edit = 1;
                else if (strcmp (optarg, "unedit") == 0)
                    the_args.unedit = 1;
                else if (strcmp (optarg, "commit") == 0)
                    the_args.commit = 1;
                else if (strcmp (optarg, "all") == 0)
                {
                    the_args.edit = 1;
                    the_args.unedit = 1;
                    the_args.commit = 1;
                }
                else if (strcmp (optarg, "none") == 0)
                {
                    the_args.edit = 0;
                    the_args.unedit = 0;
                    the_args.commit = 0;
                }
                else
                    usage (watch_usage);
                break;
            case '?':
            default:
                usage (watch_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

    if (a_omitted)
    {
        the_args.edit = 1;
        the_args.unedit = 1;
        the_args.commit = 1;
    }

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();

        if (local)
            send_arg ("-l");
        /* FIXME: copes poorly with "all" if server is extended to have
           new watch types and client is still running an old version.  */
        if (the_args.edit)
        {
            send_arg ("-a");
            send_arg ("edit");
        }
        if (the_args.unedit)
        {
            send_arg ("-a");
            send_arg ("unedit");
        }
        if (the_args.commit)
        {
            send_arg ("-a");
            send_arg ("commit");
        }
        if (!the_args.edit && !the_args.unedit && !the_args.commit)
        {
            send_arg ("-a");
            send_arg ("none");
        }
        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
        send_to_server (the_args.adding ?
                        "watch-add\012" : "watch-remove\012",
                        0);
        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    the_args.setting_default = (argc <= 0);

    lock_tree_for_write (argc, argv, local, 0);

    err = start_recursion (addremove_fileproc, addremove_filesdoneproc,
                           (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                           argc, argv, local, W_LOCAL, 0, 0, (char *)NULL,
                           1);

    Lock_Cleanup ();
```

```
    return err;
}

int
watch_add (argc, argv)
    int argc;
    char **argv;
{
    the_args.adding = 1;
    return watch_addremove (argc, argv);
}

int
watch_remove (argc, argv)
    int argc;
    char **argv;
{
    the_args.adding = 0;
    return watch_addremove (argc, argv);
}

int
watch (argc, argv)
    int argc;
    char **argv;
{
    if (argc <= 1)
        usage (watch_usage);
    if (strcmp (argv[1], "on") == 0)
    {
        --argc;
        ++argv;
        return watch_on (argc, argv);
    }
    else if (strcmp (argv[1], "off") == 0)
    {
        --argc;
        ++argv;
        return watch_off (argc, argv);
    }
    else if (strcmp (argv[1], "add") == 0)
    {
        --argc;
        ++argv;
        return watch_add (argc, argv);
    }
    else if (strcmp (argv[1], "remove") == 0)
    {
        --argc;
        ++argv;
        return watch_remove (argc, argv);
    }
    else
        usage (watch_usage);
    return 0;
}

static const char *const watchers_usage[] =
{
    "Usage: %s %s [-lR] [files...]\n",
    "\t-l\tProcess this directory only (not recursive).\n",
    "\t-R\tProcess directories recursively.\n",
    "(Specify the --help global option for a list of other help options)\n",
    NULL
};

static int watchers_fileproc PROTO ((void *callerdat,
                                     struct file_info *finfo));

static int
watchers_fileproc (callerdat, finfo)
    void *callerdat;
    struct file_info *finfo;
{
    char *them;
    char *p;

    them = fileattr_get0 (finfo->file, "_watchers");
    if (them == NULL)
        return 0;

    fputs (finfo->fullname, stdout);

    p = them;
    while (1)
    {
        putc ('\t', stdout);
        while (*p != '>' && *p != '\0')
            putc (*p++, stdout);
        if (*p == '\0')
```

```
        {
            /* Only happens if attribute is misformed.  */
            putc ('\n', stdout);
            break;
        }
        ++p;
        putc ('\t', stdout);
        while (1)
        {
            while (*p != '+' && *p != ',' && *p != '\0')
                putc (*p++, stdout);
            if (*p == '\0')
            {
                putc ('\n', stdout);
                goto out;
            }
            if (*p == ',')
            {
                ++p;
                break;
            }
            ++p;
            putc ('\t', stdout);
        }
        putc ('\n', stdout);
    }
  out:;
    return 0;
}

int
watchers (argc, argv)
    int argc;
    char **argv;
{
    int local = 0;
    int c;

    if (argc == -1)
        usage (watchers_usage);

    optind = 0;
    while ((c = getopt (argc, argv, "+lR")) != -1)
    {
        switch (c)
        {
            case 'l':
                local = 1;
                break;
            case 'R':
                local = 0;
                break;
            case '?':
            default:
                usage (watchers_usage);
                break;
        }
    }
    argc -= optind;
    argv += optind;

#ifdef CLIENT_SUPPORT
    if (client_active)
    {
        start_server ();
        ign_setup ();

        if (local)
            send_arg ("-l");
        send_file_names (argc, argv, SEND_EXPAND_WILD);
        send_files (argc, argv, local, 0, SEND_NO_CONTENTS);
        send_to_server ("watchers\012", 0);
        return get_responses_and_close ();
    }
#endif /* CLIENT_SUPPORT */

    return start_recursion (watchers_fileproc, (FILESDONEPROC) NULL,
                            (DIRENTPROC) NULL, (DIRLEAVEPROC) NULL, NULL,
                            argc, argv, local, W_LOCAL, 0, 1, (char *)NULL,
                            1);
}
```

A.66 watch.h

```
/* Interface to "cvs watch add", "cvs watchers", and related features

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

extern const char *const watch_usage[];

/* Flags to pass between the various functions making up the
   add/remove code.  All in a single structure in case there is some
   need to make the code reentrant some day.  */

struct addremove_args {
    /* A flag for each watcher type.  */
    int edit;
    int unedit;
    int commit;

    /* Are we adding or removing (non-temporary) edit,unedit,and/or commit
       watches?  */
    int adding;

    /* Should we add a temporary edit watch?  */
    int add_tedit;
    /* Should we add a temporary unedit watch?  */
    int add_tunedit;
    /* Should we add a temporary commit watch?  */
    int add_tcommit;

    /* Should we remove all temporary watches?  */
    int remove_temp;

    /* Should we set the default?  This is here for passing among various
       routines in watch.c (a good place for it if there is ever any reason
       to make the stuff reentrant), not for watch_modify_watchers.  */
    int setting_default;
};

/* Modify the watchers for FILE.  *WHAT tells what to do to them.
   If FILE is NULL, modify default args (WHAT->SETTING_DEFAULT is
   not used).  */
extern void watch_modify_watchers PROTO ((char *file,
                                          struct addremove_args *what));

extern int watch_add PROTO ((int argc, char **argv));
extern int watch_remove PROTO ((int argc, char **argv));
```

A.67 wrapper.c

```
/* This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.  */

#include "cvs.h"
#include "getline.h"

/*
 Original Author:  athanmorgan.com <Andrew C. Athan> 2/1/94
 Modified By:      vdemarcobou.shl.com

 This package was written to support the NEXTSTEP concept of
 "wrappers."  These are essentially directories that are to be
 treated as "files."  This package allows such wrappers to be
 "processed" on the way in and out of CVS.  The intended use is to
 wrap up a wrapper into a single tar, such that that tar can be
 treated as a single binary file in CVS.  To solve the problem
 effectively, it was also necessary to be able to prevent rcsmerge
 application at appropriate times.

 ---------
 Format of wrapper file ($CVSROOT/CVSROOT/cvswrappers or .cvswrappers)

 wildcard    [option value][option value]...

 where option is one of
 -f          from cvs filter       value: path to filter
 -t          to cvs filter         value: path to filter
 -m          update methodology value: MERGE or COPY
 -k          default -k rcs option to use on import or add

 and value is a single-quote delimited value.

 E.g:
 *.nib       -f 'gunzipuntar' -t 'targzip' -m 'COPY'
*/


typedef struct {
    char *wildCard;
    char *tocvsFilter;
    char *fromcvsFilter;
    char *rcsOption;
    WrapMergeMethod mergeMethod;
} WrapperEntry;

static WrapperEntry **wrap_list=NULL;
static WrapperEntry **wrap_saved_list=NULL;

static int wrap_size=0;
static int wrap_count=0;
static int wrap_tempcount=0;

/* FIXME: the relationship between wrap_count, wrap_tempcount,
 * wrap_saved_count, and wrap_saved_tempcount is not entirely clear;
 * it is certainly suspicious that wrap_saved_count is never set to a
 * value other than zero!  If the variable isn't being used, it should
 * be removed.  And in general, we should describe how temporary
 * vs. permanent wrappers are implemented, and then make sure the
 * implementation is actually doing that.
 *
 * Right now things seem to be working, but that's no guarantee there
 * isn't a bug lurking somewhere in the murk.
 */

static int wrap_saved_count=0;

static int wrap_saved_tempcount=0;

#define WRAPPER_GROW 8

void wrap_add_entry PROTO((WrapperEntry *e,int temp));
void wrap_kill PROTO((void));
void wrap_kill_temp PROTO((void));
void wrap_free_entry PROTO((WrapperEntry *e));
void wrap_free_entry_internal PROTO((WrapperEntry *e));
void wrap_restore_saved PROTO((void));

void wrap_setup()
{
    /* FIXME-reentrancy: if we do a multithreaded server, will need to
       move this to a per-connection data structure, or better yet
```

```
        think about a cleaner solution.  */
    static int wrap_setup_already_done = 0;
    char *homedir;

    if (wrap_setup_already_done != 0)
        return;
    else
        wrap_setup_already_done = 1;

#ifdef CLIENT_SUPPORT
    if (!client_active)
#endif
    {
        char *file;

        file = xmalloc (strlen (CVSroot_directory)
                        + sizeof (CVSROOTADM)
                        + sizeof (CVSROOTADM_WRAPPER)
                        + 10);
        /* Then add entries found in repository, if it exists.  */
        (void) sprintf (file, "%s/%s/%s", CVSroot_directory, CVSROOTADM,
                        CVSROOTADM_WRAPPER);
        if (isfile (file))
        {
            wrap_add_file(file,0);
        }
        free (file);
    }

    /* Then add entries found in home dir, (if user has one) and file
       exists.  */
    homedir = get_homedir ();
    if (homedir != NULL)
    {
        char *file;

        file = xmalloc (strlen (homedir) + sizeof (CVSDOTWRAPPER) + 10);
        (void) sprintf (file, "%s/%s", homedir, CVSDOTWRAPPER);
        if (isfile (file))
        {
            wrap_add_file (file, 0);
        }
        free (file);
    }

    /* FIXME: calling wrap_add() below implies that the CVSWRAPPERS
     * environment variable contains exactly one "wrapper" - a line
     * of the form
     *
     *    FILENAME_PATTERN FLAG  OPTS [ FLAG OPTS ...]
     *
     * This may disagree with the documentation, which states:
     *
     *   '$CVSWRAPPERS'
     *      A whitespace-separated list of file name patterns that CVS
     *      should treat as wrappers. *Note Wrappers::.
     *
     * Does this mean the environment variable can hold multiple
     * wrappers lines?  If so, a single call to wrap_add() is
     * insufficient.
     */

    /* Then add entries found in CVSWRAPPERS environment variable. */
    wrap_add (getenv (WRAPPER_ENV), 0);
}

#ifdef CLIENT_SUPPORT
/* Send -W arguments for the wrappers to the server.  The command must
   be one that accepts them (e.g. update, import).  */
void
wrap_send ()
{
    int i;

    for (i = 0; i < wrap_count + wrap_tempcount; ++i)
    {
        if (wrap_list[i]->tocvsFilter != NULL
            || wrap_list[i]->fromcvsFilter != NULL)
            /* For greater studliness we would print the offending option
               and (more importantly) where we found it.  */
            error (0, 0, "\
-t and -f wrapper options are not supported remotely; ignored");
        if (wrap_list[i]->mergeMethod == WRAP_COPY)
            /* For greater studliness we would print the offending option
               and (more importantly) where we found it.  */
            error (0, 0, "\
-m wrapper option is not supported remotely; ignored");
        if (wrap_list[i]->rcsOption != NULL)
        {
            send_to_server ("Argument -W\012Argument ", 0);
```

```
            send_to_server (wrap_list[i]->wildCard, 0);
            send_to_server (" -k '", 0);
            send_to_server (wrap_list[i]->rcsOption, 0);
            send_to_server ("'\012", 0);
        }
    }
}
#endif /* CLIENT_SUPPORT */

#if defined(SERVER_SUPPORT) || defined(CLIENT_SUPPORT)
/* Output wrapper entries in the format of cvswrappers lines.
 *
 * This is useful when one side of a client/server connection wants to
 * send its wrappers to the other; since the receiving side would like
 * to use wrap_add() to incorporate the wrapper, it's best if the
 * entry arrives in this format.
 *
 * The entries are stored in 'line', which is allocated here.  Caller
 * can free() it.
 *
 * If first_call_p is nonzero, then start afresh.  */
void
wrap_unparse_rcs_options (line, first_call_p)
    char **line;
    int first_call_p;
{
    /* FIXME-reentrancy: we should design a reentrant interface, like
       a callback which gets handed each wrapper (a multithreaded
       server being the most concrete reason for this, but the
       non-reentrant interface is fairly unnecessary/ugly).  */
    static int i;

    if (first_call_p)
        i = 0;

    for (; i < wrap_count + wrap_tempcount; ++i)
    {
        if (wrap_list[i]->rcsOption != NULL)
        {
            *line = xmalloc (strlen (wrap_list[i]->wildCard)
                             + strlen ("\t")
                             + strlen (" -k '")
                             + strlen (wrap_list[i]->rcsOption)
                             + strlen ("'")
                             + 1);  /* leave room for '\0' */

            strcpy (*line, wrap_list[i]->wildCard);
            strcat (*line, " -k '");
            strcat (*line, wrap_list[i]->rcsOption);
            strcat (*line, "'");

            /* We're going to miss the increment because we return, so
               do it by hand. */
            ++i;

            return;
        }
    }

    *line = NULL;
    return;
}
#endif /* SERVER_SUPPORT || CLIENT_SUPPORT */

/*
 * Open a file and read lines, feeding each line to a line parser. Arrange
 * for keeping a temporary list of wrappers at the end, if the "temp"
 * argument is set.
 */
void
wrap_add_file (file, temp)
    const char *file;
    int temp;
{
    FILE *fp;
    char *line = NULL;
    size_t line_allocated = 0;

    wrap_restore_saved ();
    wrap_kill_temp ();

    /* Load the file.  */
    fp = CVS_FOPEN (file, "r");
    if (fp == NULL)
    {
        if (!existence_error (errno))
            error (0, errno, "cannot open %s", file);
        return;
    }
    while (getline (&line, &line_allocated, fp) >= 0)
```

```
        wrap_add (line, temp);
    if (line)
        free (line);
    if (ferror (fp))
        error (0, errno, "cannot read %s", file);
    if (fclose (fp) == EOF)
        error (0, errno, "cannot close %s", file);
}

void
wrap_kill()
{
    wrap_kill_temp();
    while(wrap_count)
        wrap_free_entry(wrap_list[--wrap_count]);
}

void
wrap_kill_temp()
{
    WrapperEntry **temps=wrap_list+wrap_count;

    while(wrap_tempcount)
        wrap_free_entry(temps[--wrap_tempcount]);
}

void
wrap_free_entry(e)
     WrapperEntry *e;
{
    wrap_free_entry_internal(e);
    free(e);
}

void
wrap_free_entry_internal(e)
    WrapperEntry *e;
{
    free (e->wildCard);
    if (e->tocvsFilter)
        free (e->tocvsFilter);
    if (e->fromcvsFilter)
        free (e->fromcvsFilter);
    if (e->rcsOption)
        free (e->rcsOption);
}

void
wrap_restore_saved()
{
    if(!wrap_saved_list)
        return;

    wrap_kill();

    free(wrap_list);

    wrap_list=wrap_saved_list;
    wrap_count=wrap_saved_count;
    wrap_tempcount=wrap_saved_tempcount;

    wrap_saved_list=NULL;
    wrap_saved_count=0;
    wrap_saved_tempcount=0;
}

void
wrap_add (line, isTemp)
   char *line;
   int          isTemp;
{
    char *temp;
    char ctemp;
    WrapperEntry e;
    char opt;

    if (!line || line[0] == '#')
        return;

    memset (&e, 0, sizeof(e));

        /* Search for the wild card */
    while(*line && isspace(*line))
        ++line;
    for(temp=line;*line && !isspace(*line);++line)
        ;
    if(temp==line)
        return;

    ctemp=*line;
```

```
    *line='\0';

    e.wildCard=xstrdup(temp);
    *line=ctemp;

    while(*line){
        /* Search for the option */
        while(*line && *line!='-')
            ++line;
        if(!*line)
            break;
        ++line;
        if(!*line)
            break;
        opt=*line;

        /* Search for the filter commandline */
        for(++line;*line && *line!='\'';++line);
        if(!*line)
            break;

        for(temp=++line;*line && (*line!='\'' || line[-1]=='\\');++line)
            ;

        /* This used to "break;" (ignore the option) if there was a
           single character between the single quotes (I'm guessing
           that was accidental).  Now it "break;"s if there are no
           characters.  I'm not sure either behavior is particularly
           necessary--the current options might not require "
           arguments, but surely some future option legitimately
           might.  Also I'm not sure that ignoring the option is a
           swift way to handle syntax errors in general.  */
        if (line==temp)
            break;

        ctemp=*line;
        *line='\0';
        switch(opt){
        case 'f':
            /* Before this is reenabled, need to address the problem in
               commit.c (see http://www.cyclic.com/cvs/dev-wrap.txt).  */
            error (1, 0,
                   "-t/-f wrappers not supported by this version of CVS");

            if(e.fromcvsFilter)
                free(e.fromcvsFilter);
            /* FIXME: error message should say where the bad value
               came from.  */
            e.fromcvsFilter=expand_path (temp, "<wrapper>", 0);
            if (!e.fromcvsFilter)
                error (1, 0, "Correct above errors first");
            break;
        case 't':
            /* Before this is reenabled, need to address the problem in
               commit.c (see http://www.cyclic.com/cvs/dev-wrap.txt).  */
            error (1, 0,
                   "-t/-f wrappers not supported by this version of CVS");

            if(e.tocvsFilter)
                free(e.tocvsFilter);
            /* FIXME: error message should say where the bad value
               came from.  */
            e.tocvsFilter=expand_path (temp, "<wrapper>", 0);
            if (!e.tocvsFilter)
                error (1, 0, "Correct above errors first");
            break;
        case 'm':
            /* FIXME: look into whether this option is still relevant given
               the 24 Jun 96 change to merge_file.  */
            if(*temp=='C' || *temp=='c')
                e.mergeMethod=WRAP_COPY;
            else
                e.mergeMethod=WRAP_MERGE;
            break;
        case 'k':
            if (e.rcsOption)
                free (e.rcsOption);
            e.rcsOption = xstrdup (temp);
            break;
        default:
            break;
        }
        *line=ctemp;
        if(!*line)break;
        ++line;
    }

    wrap_add_entry(&e, isTemp);
}
```

```
void
wrap_add_entry(e, temp)
    WrapperEntry *e;
    int temp;
{
    int x;
    if(wrap_count+wrap_tempcount>=wrap_size){
        wrap_size += WRAPPER_GROW;
        wrap_list = (WrapperEntry **) xrealloc ((char *) wrap_list,
                                                wrap_size *
                                                sizeof (WrapperEntry *));
    }

    if(!temp && wrap_tempcount){
        for(x=wrap_count+wrap_tempcount-1;x>=wrap_count;--x)
            wrap_list[x+1]=wrap_list[x];
    }

    x=(temp ? wrap_count+(wrap_tempcount++):(wrap_count++));
    wrap_list[x]=(WrapperEntry *)xmalloc(sizeof(WrapperEntry));
    wrap_list[x]->wildCard=e->wildCard;
    wrap_list[x]->fromcvsFilter=e->fromcvsFilter;
    wrap_list[x]->tocvsFilter=e->tocvsFilter;
    wrap_list[x]->mergeMethod=e->mergeMethod;
    wrap_list[x]->rcsOption = e->rcsOption;
}

/* Return 1 if the given filename is a wrapper filename */
int
wrap_name_has (name,has)
    const char   *name;
    WrapMergeHas  has;
{
    int x,count=wrap_count+wrap_tempcount;
    char *temp;

    for(x=0;x<count;++x)
        if (CVS_FNMATCH (wrap_list[x]->wildCard, name, 0) == 0){
            switch(has){
            case WRAP_TOCVS:
                temp=wrap_list[x]->tocvsFilter;
                break;
            case WRAP_FROMCVS:
                temp=wrap_list[x]->fromcvsFilter;
                break;
            case WRAP_RCSOPTION:
                temp = wrap_list[x]->rcsOption;
                break;
            default:
                abort ();
            }
            if(temp==NULL)
                return (0);
            else
                return (1);
        }
    return (0);
}

static WrapperEntry *wrap_matching_entry PROTO ((const char *));

static WrapperEntry *
wrap_matching_entry (name)
    const char *name;
{
    int x,count=wrap_count+wrap_tempcount;

    for(x=0;x<count;++x)
        if (CVS_FNMATCH (wrap_list[x]->wildCard, name, 0) == 0)
            return wrap_list[x];
    return (WrapperEntry *)NULL;
}

/* Return the RCS options for FILENAME in a newly malloc'd string.  If
   ASFLAG, then include "-k" at the beginning (e.g. "-kb"), otherwise
   just give the option itself (e.g. "b").  */
char *
wrap_rcsoption (filename, asflag)
    const char *filename;
    int asflag;
{
    WrapperEntry *e = wrap_matching_entry (filename);
    char *buf;

    if (e == NULL || e->rcsOption == NULL || (*e->rcsOption == '\0'))
        return NULL;

    buf = xmalloc (strlen (e->rcsOption) + 3);
    if (asflag)
    {
```

```
        strcpy (buf, "-k");
        strcat (buf, e->rcsOption);
    }
    else
    {
        strcpy (buf, e->rcsOption);
    }
    return buf;
}

char *
wrap_tocvs_process_file(fileName)
    const char *fileName;
{
    WrapperEntry *e=wrap_matching_entry(fileName);
    static char *buf = NULL;
    char *args;

    if(e==NULL || e->tocvsFilter==NULL)
        return NULL;

    if (buf != NULL)
        free (buf);
    buf = cvs_temp_name ();

    args = xmalloc (strlen (e->tocvsFilter)
                    + strlen (fileName)
                    + strlen (buf));
    /* FIXME: sprintf will blow up if the format string contains items other
       than %s, or contains too many %s's.  We should instead be parsing
       e->tocvsFilter ourselves and giving a real error.  */
    sprintf (args, e->tocvsFilter, fileName, buf);
    run_setup (args);
    run_exec(RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL|RUN_REALLY );
    free (args);

    return buf;
}

int
wrap_merge_is_copy (fileName)
    const char *fileName;
{
    WrapperEntry *e=wrap_matching_entry(fileName);
    if(e==NULL || e->mergeMethod==WRAP_MERGE)
        return 0;

    return 1;
}

void
wrap_fromcvs_process_file(fileName)
    const char *fileName;
{
    char *args;
    WrapperEntry *e=wrap_matching_entry(fileName);

    if(e==NULL || e->fromcvsFilter==NULL)
        return;

    args = xmalloc (strlen (e->fromcvsFilter)
                    + strlen (fileName));
    /* FIXME: sprintf will blow up if the format string contains items other
       than %s, or contains too many %s's.  We should instead be parsing
       e->fromcvsFilter ourselves and giving a real error.  */
    sprintf (args, e->fromcvsFilter, fileName);
    run_setup (args);
    run_exec(RUN_TTY, RUN_TTY, RUN_TTY, RUN_NORMAL );
    free (args);
    return;
}
```

A.68 zlib.c

```
/* zlib.c — interface to the zlib compression library
   Ian Lance Taylor <iancygnus.com>

   This file is part of GNU CVS.

   GNU CVS is free software; you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published by the
   Free Software Foundation; either version 2, or (at your option) any
   later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*/

/* The routines in this file are the interface between the CVS
   client/server support and the zlib compression library.  */

#include <assert.h>
#include "cvs.h"
#include "buffer.h"

#if defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT)

#include "zlib.h"

/* OS/2 doesn't have EIO.  FIXME: this whole notion of turning
   a different error into EIO strikes me as pretty dubious.  */
#if !defined (EIO)
#define EIO EBADPOS
#endif

/* The compression interface is built upon the buffer data structure.
   We provide a buffer type which compresses or decompresses the data
   which passes through it.  An input buffer decompresses the data
   read from an underlying buffer, and an output buffer compresses the
   data before writing it to an underlying buffer.  */

/* This structure is the closure field of the buffer.  */

struct compress_buffer
{
   /* The underlying buffer.  */
   struct buffer *buf;
   /* The compression information.  */
   z_stream zstr;
};

static void compress_error PROTO((int, int, z_stream *, const char *));
static int compress_buffer_input PROTO((void *, char *, int, int, int *));
static int compress_buffer_output PROTO((void *, const char *, int, int *));
static int compress_buffer_flush PROTO((void *));
static int compress_buffer_block PROTO((void *, int));
static int compress_buffer_shutdown_input PROTO((void *));
static int compress_buffer_shutdown_output PROTO((void *));

/* Report an error from one of the zlib functions.  */

static void
compress_error (status, zstatus, zstr, msg)
    int status;
    int zstatus;
    z_stream *zstr;
    const char *msg;
{
   int hold_errno;
   const char *zmsg;
   char buf[100];

   hold_errno = errno;

   zmsg = zstr->msg;
   if (zmsg == NULL)
   {
      sprintf (buf, "error %d", zstatus);
      zmsg = buf;
   }

   error (status,
          zstatus == Z_ERRNO ? hold_errno : 0,
          "%s: %s", msg, zmsg);
}

/* Create a compression buffer.  */

struct buffer *
compress_buffer_initialize (buf, input, level, memory)
    struct buffer *buf;
```

```
     int input;
     int level;
     void (*memory) PROTO((struct buffer *));
{
  struct compress_buffer *n;
  int zstatus;

  n = (struct compress_buffer *) xmalloc (sizeof *n);
  memset (n, 0, sizeof *n);

  n->buf = buf;

  if (input)
    zstatus = inflateInit (&n->zstr);
  else
    zstatus = deflateInit (&n->zstr, level);
  if (zstatus != Z_OK)
    compress_error (1, zstatus, &n->zstr, "compression initialization");

  /* There may already be data buffered on BUF.  For an output
     buffer, this is OK, because these routines will just use the
     buffer routines to append data to the (uncompressed) data
     already on BUF.  An input buffer expects to handle a single
     buffer_data of buffered input to be uncompressed, so that is OK
     provided there is only one buffer.  At present that is all
     there ever will be; if this changes, compress_buffer_input must
     be modified to handle multiple input buffers.  */
  assert (! input || buf->data == NULL || buf->data->next == NULL);

  return buf_initialize (input ? compress_buffer_input : NULL,
                         input ? NULL : compress_buffer_output,
                         input ? NULL : compress_buffer_flush,
                         compress_buffer_block,
                         (input
                          ? compress_buffer_shutdown_input
                          : compress_buffer_shutdown_output),
                         memory,
                         n);
}

/* Input data from a compression buffer.  */

static int
compress_buffer_input (closure, data, need, size, got)
     void *closure;
     char *data;
     int need;
     int size;
     int *got;
{
  struct compress_buffer *cb = (struct compress_buffer *) closure;
  struct buffer_data *bd;

  if (cb->buf->input == NULL)
    abort ();

  /* We use a single buffer_data structure to buffer up data which
     the z_stream structure won't use yet.  We can safely store this
     on cb->buf->data, because we never call the buffer routines on
     cb->buf; we only call the buffer input routine, since that
     gives us the semantics we want.  As noted in
     compress_buffer_initialize, the buffer_data structure may
     already exist, and hold data which was already read and
     buffered before the decompression began.  */
  bd = cb->buf->data;
  if (bd == NULL)
  {
    bd = ((struct buffer_data *) malloc (sizeof (struct buffer_data)));
    if (bd == NULL)
      return -2;
    bd->text = (char *) malloc (BUFFER_DATA_SIZE);
    if (bd->text == NULL)
    {
      free (bd);
      return -2;
    }
    bd->bufp = bd->text;
    bd->size = 0;
    cb->buf->data = bd;
  }

  cb->zstr.avail_out = size;
  cb->zstr.next_out = (Bytef *) data;

  while (1)
  {
    int zstatus, sofar, status, nread;

    /* First try to inflate any data we already have buffered up.
       This is useful even if we don't have any buffered data,
```

```
            because there may be data buffered inside the z_stream
            structure.  */

          cb->zstr.avail_in = bd->size;
          cb->zstr.next_in = (Bytef *) bd->bufp;

          do
          {
              zstatus = inflate (&cb->zstr, Z_NO_FLUSH);
              if (zstatus == Z_STREAM_END)
                  break;
              if (zstatus != Z_OK && zstatus != Z_BUF_ERROR)
              {
                  compress_error (0, zstatus, &cb->zstr, "inflate");
                  return EIO;
              }
          } while (cb->zstr.avail_in > 0
                   && cb->zstr.avail_out > 0);

          bd->size = cb->zstr.avail_in;
          bd->bufp = (char *) cb->zstr.next_in;

          if (zstatus == Z_STREAM_END)
              return -1;

          /* If we have obtained NEED bytes, then return, unless NEED is
             zero and we haven't obtained anything at all.  If NEED is
             zero, we will keep reading from the underlying buffer until
             we either can't read anything, or we have managed to
             inflate at least one byte.  */
          sofar = size - cb->zstr.avail_out;
          if (sofar > 0 && sofar >= need)
              break;

          /* All our buffered data should have been processed at this
             point.  */
          assert (bd->size == 0);

          /* This will work well in the server, because this call will
             do an unblocked read and fetch all the available data.  In
             the client, this will read a single byte from the stdio
             stream, which will cause us to call inflate once per byte.
             It would be more efficient if we could make a call which
             would fetch all the available bytes, and at least one byte.  */

          status = (*cb->buf->input) (cb->buf->closure, bd->text,
                                      need > 0 ? 1 : 0,
                                      BUFFER_DATA_SIZE, &nread);
          if (status != 0)
              return status;

          /* If we didn't read anything, then presumably the buffer is
             in nonblocking mode, and we should just get out now with
             whatever we've inflated.  */
          if (nread == 0)
          {
              assert (need == 0);
              break;
          }

          bd->bufp = bd->text;
          bd->size = nread;
      }

      *got = size - cb->zstr.avail_out;

      return 0;
  }

  /* Output data to a compression buffer.  */

  static int
  compress_buffer_output (closure, data, have, wrote)
       void *closure;
       const char *data;
       int have;
       int *wrote;
  {
      struct compress_buffer *cb = (struct compress_buffer *) closure;

      cb->zstr.avail_in = have;
      cb->zstr.next_in = (unsigned char *) data;

      while (cb->zstr.avail_in > 0)
      {
          char buffer[BUFFER_DATA_SIZE];
          int zstatus;

          cb->zstr.avail_out = BUFFER_DATA_SIZE;
          cb->zstr.next_out = (unsigned char *) buffer;
```

```
270         zstatus = deflate (&cb->zstr, Z_NO_FLUSH);
            if (zstatus != Z_OK)
            {
                compress_error (0, zstatus, &cb->zstr, "deflate");
                return EIO;
            }

            if (cb->zstr.avail_out != BUFFER_DATA_SIZE)
                buf_output (cb->buf, buffer,
                            BUFFER_DATA_SIZE - cb->zstr.avail_out);
280     }

        *wrote = have;

        /* We will only be here because buf_send_output was called on the
           compression buffer.  That means that we should now call
           buf_send_output on the underlying buffer.  */
        return buf_send_output (cb->buf);
    }

290 /* Flush a compression buffer.  */

    static int
    compress_buffer_flush (closure)
         void *closure;
    {
        struct compress_buffer *cb = (struct compress_buffer *) closure;

        cb->zstr.avail_in = 0;
        cb->zstr.next_in = NULL;
300
        while (1)
        {
            char buffer[BUFFER_DATA_SIZE];
            int zstatus;

            cb->zstr.avail_out = BUFFER_DATA_SIZE;
            cb->zstr.next_out = (unsigned char *) buffer;

            zstatus = deflate (&cb->zstr, Z_SYNC_FLUSH);
310
            /* The deflate function will return Z_BUF_ERROR if it can't do
               anything, which in this case means that all data has been
               flushed.  */
            if (zstatus == Z_BUF_ERROR)
                break;

            if (zstatus != Z_OK)
            {
                compress_error (0, zstatus, &cb->zstr, "deflate flush");
320             return EIO;
            }

            if (cb->zstr.avail_out != BUFFER_DATA_SIZE)
                buf_output (cb->buf, buffer,
                            BUFFER_DATA_SIZE - cb->zstr.avail_out);

            /* If the deflate function did not fill the output buffer,
               then all data has been flushed.  */
            if (cb->zstr.avail_out > 0)
330             break;
        }

        /* Now flush the underlying buffer.  Note that if the original
           call to buf_flush passed 1 for the BLOCK argument, then the
           buffer will already have been set into blocking mode, so we
           should always pass 0 here.  */
        return buf_flush (cb->buf, 0);
    }

340 /* The block routine for a compression buffer.  */

    static int
    compress_buffer_block (closure, block)
         void *closure;
         int block;
    {
        struct compress_buffer *cb = (struct compress_buffer *) closure;

        if (block)
350         return set_block (cb->buf);
        else
            return set_nonblock (cb->buf);
    }

    /* Shut down an input buffer.  */

    static int
    compress_buffer_shutdown_input (closure)
```

```
     void *closure;
{
  struct compress_buffer *cb = (struct compress_buffer *) closure;
  int zstatus;

  /* Pick up any trailing data, such as the checksum.  */
  while (1)
  {
    int status, nread;
    char buf[100];

    status = compress_buffer_input (cb, buf, 0, sizeof buf, &nread);
    if (status == -1)
      break;
    if (status != 0)
      return status;
  }

  zstatus = inflateEnd (&cb->zstr);
  if (zstatus != Z_OK)
  {
    compress_error (0, zstatus, &cb->zstr, "inflateEnd");
    return EIO;
  }

  return buf_shutdown (cb->buf);
}

/* Shut down an output buffer.  */

static int
compress_buffer_shutdown_output (closure)
     void *closure;
{
  struct compress_buffer *cb = (struct compress_buffer *) closure;
  int zstatus, status;

  do
  {
    char buffer[BUFFER_DATA_SIZE];

    cb->zstr.avail_out = BUFFER_DATA_SIZE;
    cb->zstr.next_out = (unsigned char *) buffer;

    zstatus = deflate (&cb->zstr, Z_FINISH);
    if (zstatus != Z_OK && zstatus != Z_STREAM_END)
    {
      compress_error (0, zstatus, &cb->zstr, "deflate finish");
      return EIO;
    }

    if (cb->zstr.avail_out != BUFFER_DATA_SIZE)
      buf_output (cb->buf, buffer,
                  BUFFER_DATA_SIZE - cb->zstr.avail_out);
  } while (zstatus != Z_STREAM_END);

  zstatus = deflateEnd (&cb->zstr);
  if (zstatus != Z_OK)
  {
    compress_error (0, zstatus, &cb->zstr, "deflateEnd");
    return EIO;
  }

  status = buf_flush (cb->buf, 1);
  if (status != 0)
    return status;

  return buf_shutdown (cb->buf);
}


/* Here is our librarified gzip implementation.  It is very minimal
   but attempts to be RFC1952 compliant.  */
/* Note that currently only the client uses the gzip library.  If we
   make the server use it too (which should be straightforward), then
   filter_stream_through_program, filter_through_gzip, and
   filter_through_gunzip can go away.  */

/* BUF should contain SIZE bytes of gzipped data (RFC1952/RFC1951).
   We are to uncompress the data and write the result to the file
   descriptor FD.  If something goes wrong, give an error message
   mentioning FULLNAME as the name of the file for FD (and make it a
   fatal error if we can't recover from it).  */

void
gunzip_and_write (fd, fullname, buf, size)
  int fd;
  char *fullname;
  unsigned char *buf;
```

```
    size_t size;
{
    size_t pos;
    z_stream zstr;
    int zstatus;
    unsigned char outbuf[32768];
    unsigned long crc;

    if (buf[0] != 31 || buf[1] != 139)
        error (1, 0, "gzipped data does not start with gzip identification");
    if (buf[2] != 8)
        error (1, 0, "only the deflate compression method is supported");

    /* Skip over the fixed header, and then skip any of the variable-length
       fields.  */
    pos = 10;
    if (buf[3] & 4)
        pos += buf[pos] + (buf[pos + 1] << 8) + 2;
    if (buf[3] & 8)
        pos += strlen (buf + pos) + 1;
    if (buf[3] & 16)
        pos += strlen (buf + pos) + 1;
    if (buf[3] & 2)
        pos += 2;

    memset (&zstr, 0, sizeof zstr);
    /* Passing a negative argument tells zlib not to look for a zlib
       (RFC1950) header.  This is an undocumented feature; I suppose if
       we wanted to be anal we could synthesize a header instead,
       but why bother?  */
    zstatus = inflateInit2 (&zstr, -15);

    if (zstatus != Z_OK)
        compress_error (1, zstatus, &zstr, fullname);

    /* I don't see why we should have to include the 8 byte trailer in
       avail_in.  But I see that zlib/gzio.c does, and it seemed to fix
       a fairly rare bug in which we'd get a Z_BUF_ERROR for no obvious
       reason.  */
    zstr.avail_in = size - pos;
    zstr.next_in = buf + pos;

    crc = crc32 (0, NULL, 0);

    do
    {
        zstr.avail_out = sizeof (outbuf);
        zstr.next_out = outbuf;
        zstatus = inflate (&zstr, Z_NO_FLUSH);
        if (zstatus != Z_STREAM_END && zstatus != Z_OK)
            compress_error (1, zstatus, &zstr, fullname);
        if (write (fd, outbuf, sizeof (outbuf) - zstr.avail_out) < 0)
            error (1, errno, "writing decompressed file %s", fullname);
        crc = crc32 (crc, outbuf, sizeof (outbuf) - zstr.avail_out);
    } while (zstatus != Z_STREAM_END);
    zstatus = inflateEnd (&zstr);
    if (zstatus != Z_OK)
        compress_error (0, zstatus, &zstr, fullname);

    if (crc != (buf[zstr.total_in + 10]
                + (buf[zstr.total_in + 11] << 8)
                + (buf[zstr.total_in + 12] << 16)
                + (buf[zstr.total_in + 13] << 24)))
        error (1, 0, "CRC error uncompressing %s", fullname);

    if (zstr.total_out != (buf[zstr.total_in + 14]
                           + (buf[zstr.total_in + 15] << 8)
                           + (buf[zstr.total_in + 16] << 16)
                           + (buf[zstr.total_in + 17] << 24)))
        error (1, 0, "invalid length uncompressing %s", fullname);
}

/* Read all of FD and put the gzipped data (RFC1952/RFC1951) into *BUF,
   replacing previous contents of *BUF.  *BUF is malloc'd and *SIZE is
   its allocated size.  Put the actual number of bytes of data in
   *LEN.  If something goes wrong, give an error message mentioning
   FULLNAME as the name of the file for FD (and make it a fatal error
   if we can't recover from it).  LEVEL is the compression level (1-9).  */

void
read_and_gzip (fd, fullname, buf, size, len, level)
    int fd;
    char *fullname;
    unsigned char **buf;
    size_t *size;
    size_t *len;
    int level;
{
    z_stream zstr;
    int zstatus;
```

```
    unsigned char inbuf[8192];
    int nread;
    unsigned long crc;

    if (*size < 1024)
    {
        *size = 1024;
        *buf = (unsigned char *) xrealloc (*buf, *size);
    }
    (*buf)[0] = 31;
    (*buf)[1] = 139;
    (*buf)[2] = 8;
    (*buf)[3] = 0;
    (*buf)[4] = (*buf)[5] = (*buf)[6] = (*buf)[7] = 0;
    /* Could set this based on level, but why bother?  */
    (*buf)[8] = 0;
    (*buf)[9] = 255;

    memset (&zstr, 0, sizeof zstr);
    zstatus = deflateInit2 (&zstr, level, Z_DEFLATED, -15, 8,
                            Z_DEFAULT_STRATEGY);
    crc = crc32 (0, NULL, 0);
    if (zstatus != Z_OK)
        compress_error (1, zstatus, &zstr, fullname);
    zstr.avail_out = *size;
    zstr.next_out = *buf + 10;

    while (1)
    {
        int finish = 0;

        nread = read (fd, inbuf, sizeof inbuf);
        if (nread < 0)
            error (1, errno, "cannot read %s", fullname);
        else if (nread == 0)
            /* End of file.  */
            finish = 1;
        crc = crc32 (crc, inbuf, nread);
        zstr.next_in = inbuf;
        zstr.avail_in = nread;

        do
        {
            size_t offset;

            /* I don't see this documented anywhere, but deflate seems
               to tend to dump core sometimes if we pass it Z_FINISH and
               a small (e.g. 2147 byte) avail_out.  So we insist on at
               least 4096 bytes (that is what zlib/gzio.c uses).  */

            if (zstr.avail_out < 4096)
            {
                offset = zstr.next_out - *buf;
                *size *= 2;
                *buf = xrealloc (*buf, *size);
                zstr.next_out = *buf + offset;
                zstr.avail_out = *size - offset;
            }

            zstatus = deflate (&zstr, finish ? Z_FINISH : 0);
            if (zstatus == Z_STREAM_END)
                goto done;
            else if (zstatus != Z_OK)
                compress_error (0, zstatus, &zstr, fullname);
        } while (zstr.avail_out == 0);
    }
 done:
    *(*buf + zstr.total_out + 10) = crc & 0xff;
    *(*buf + zstr.total_out + 11) = (crc >> 8) & 0xff;
    *(*buf + zstr.total_out + 12) = (crc >> 16) & 0xff;
    *(*buf + zstr.total_out + 13) = (crc >> 24) & 0xff;

    *(*buf + zstr.total_out + 14) = zstr.total_in & 0xff;
    *(*buf + zstr.total_out + 15) = (zstr.total_in >> 8) & 0xff;
    *(*buf + zstr.total_out + 16) = (zstr.total_in >> 16) & 0xff;
    *(*buf + zstr.total_out + 17) = (zstr.total_in >> 24) & 0xff;

    *len = zstr.total_out + 18;

    zstatus = deflateEnd (&zstr);
    if (zstatus != Z_OK)
        compress_error (0, zstatus, &zstr, fullname);
}
#endif /* defined (SERVER_SUPPORT) || defined (CLIENT_SUPPORT) */
```

Bibliography

[1] Karl Franz Fogel. *Open Source Development with CVS*. The Coriolis Group, 1999.

[2] Tim Mikkelsen, Suzanne Pherigo. *Practical Software Configuration Management: The Latenight Developer's Handbook*. Prentice Hall, 1997.

[3] Per Cederquist et al. *CVS – Concurrent Versions System.* <http://www.loria.fr/~molli/cvs/doc/cvs_toc.html>.

[4] Ram Rajadhyaksha et al. *MacCVS Pro* <http://www.maccvs.org/>, <http://sourceforge.net/projects/maccvspro/>