Symbolics Internal Guide to Version Control

The Version Control Regime

Introduction to Version Control

This Document

This document describes the design of the present version control regime. The version control regime is presented in two parts:

- A description of the components of the version control regime and how they fit together
- A description of how to use the version control tools

The Readership of this Document

This document is addressed to programmers at Symbolics who are assumed to be familiar with the Symbolics-Lisp language and environment. At the present time, this is an internal document and is not part of the distributed Symbolics Documentation Set.

Note: Throughout the document, the ⊗ symbol indicates a feature that is not yet implemented.

Why Do We Need Version Control?

Presently (i.e., prior to the widespread use of version control), all sources of programs and documents are stored in unstructured text files. Each of these text files contains the source for a large number of objects. Modifications to these sources leave no record of the author, time, or reason for the modifications to individual objects.

This situation results in two major hindrances to software development. First, the lack of historical information forces people to spend time figuring out the reason for changes. Second, there is no support for organizing parallel development projects. That is, when two (or more) people make divergent versions of the system (called *branches*), the conflicts between the branches must be found and resolved by exhaustive comparison of all of the sources involved. This is very labor-intensive.

The version control project addresses these problems by making two major changes:

- Use of a new storage format for source files that maintains a change history

- Development of a set of tools that use these files to keep orderly records of source changes

The tools for keeping orderly records help solve the problem of tracing the history of a parallel development branch. They also do much of the work of merging and reconciling parallel versions. They assist in migrating changes back and forth between versions.

Another benefit of version control is the way it compactly stores source files. In the past, multiple versions of the same code could soak up a great deal of disk space. Version control solves the disk storage problem by using a much more compact representation of source files; only the differences between subsequent versions are stored.

The Components of Version Control

Version-controlled Files

A version-controlled file contains source text edited by a user. The current version control regime supports Lisp source. In principle, any sort of text file could be supported, but we haven't qualified with anything other than Lisp.

A version-controlled file is subdivided into a collection of *hard sections,* each of which contains a "logical unit" of source code. Since the actual meaning of "logical unit" is left to the programmer, we employ the term loosely here. Typical, though, *definitions* are used as logical units.

A *file version image* is a version of the hard sections in a version-controlled file. In the following example, there are two file version images, **initial.0** and **initial.1** in a version-controlled file. Programmers see only the file version image they want to see. Each file version image contains a collection of hard sections. Here the hard sections are **foo, bar,** and **baz.**

```
What the              What's actually
user sees:            in the file:

Initial.1             Initial.0
-------               ------
foo                   foo
-------               ------
bar                   bar
-------               ------
baz                   baz
-------               ------

                      Initial.1
                      ------
                      foo
                      ------
                      bar
                      ------
                      baz
                      ------
```

Hard Sections

Each version-controlled file is divided into a series of hard sections. Each hard section contains a "logical unit" of source text, typically a definition. Since Zmacs is not in general capable of detecting logical unit boundaries, you must explicitly make new hard section boundaries with a Zmacs command when you add new units to a file.

Natural units of information must not be split across hard section boundaries. For example, in Lisp code, a top level form cannot be split across hard section boundaries.

Hard sections do not have names. However, Zmacs tries to insure that a given hard section always contains the same "definition," within the Symbolics-Lisp paradigm of Function Spec and Definition Type. The existing Zmacs heuristic sectionization algorithm is used within each hard section to locate definitions.

Hard sections are displayed with title lines obtained using the paradigm of Function Spec and Definition Type. The interval label is displayed within a border at the top of the hard section.

Lines and Branches

Lines

The result of successive editing sessions on a file version image is called a *line*. As mentioned, a file version image contains a version of a collection of related hard sections. This line can be visualized as in the example below.

```
initial.0
 |
 |
initial.1
 |
 |
 .
 .
 .
initial.n
```

Here **initial.0** is the first file version image, and **initial.n** is the last file version image. When editing a version-controlled file, you can display any version in the line.

Branches

Parallel development is represented in version control files by splitting the line of versions into multiple lines, called *branches*. The point at which the branch occurs is called the *branchpoint*.

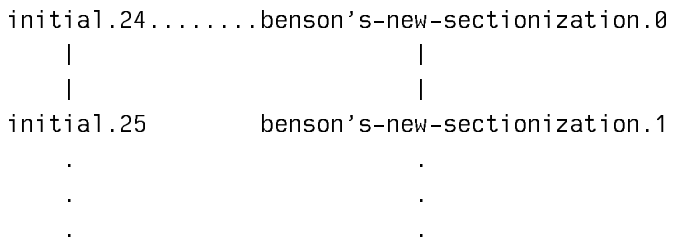
Branches have names, and the name of a branch is shown in the window legend line of a Zmacs buffer (the bottom line of the buffer, on top of the text window border).

In the next figure, **a-foo** is the name of a branch of a version controlled file. Version **a-foo.1** was modified to create a branchpoint and form the branch **b-foo**. **b-foo.0** is the first version of the new branch.

```
a-foo.0
   |
   |
a-foo.1 ..................b.foo.0
   |                         |
   |                         |
a-foo.2                  b-foo.1
   .                         .
   .                         .
   .                         .
a-foo.n                  b-foo.n
```

You make a branch when you need to save away one or more changed hard sections without making the changes visible to all of the other people using or developing the sources. For example, if you were making a minor change to the system over a period of several days, you would create a branch and save intermediate versions as changes along the branch.

The next example diagram shows an individual developer making a branch to a piece of the Zwei system because the new versions probably won't work, and the developer wants to keep them separate from the existing, working source code



Development of both branches **initial** and **benson's-new-sectionization** can proceed simultaneously.

Branches are also used for major development projects taken on by groups of programmers. For example, if version control had been available when the ADSI character object work started, the developers would have started a new branch for their changes.

Merging Branches

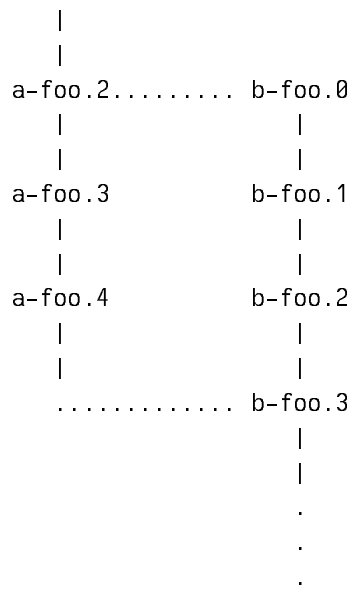
To merge two branches is to reconcile the differences between them. Merging is a directed operation: at the end of it, one of the two branches reflects the changes in both, and the other is unchanged. In other words, the changes in one branch (the source) are *merged into* the other branch (the target).

Here are some examples of circumstances when you use merging:

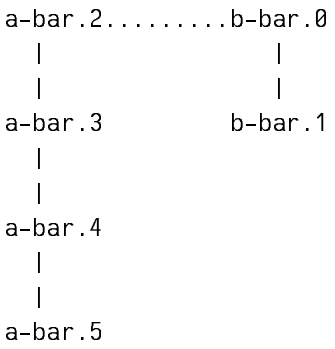
- You have been developing a change in a branch off of the main line of development. You are ready to include the change in the main line. You merge the changes in your branch into the main branch.

- You are developing a major change to the system over a period of time. You have your version of the source in a branch of one or more source files. Every so often, you want to pick up any changes to those files in the main line of development and merge them into your version, so that you have less work to do later.

In the next figure a line of changes branches at **a-foo.2** and merges at **b-foo.3.**



Not all branches merge. Some branches turn out to be dead ends in the development process, as the example below shows:



Here **b-bar.1** was abandoned as the last version of branch **b-bar.**

Disposition of a Branch

⊗ *This section describes features that are not yet available*

When you are done with a branch, you have several choices as to the disposition of its contents.

You can choose to remove some or all of the branch history from the file. You can *compress* the branch, removing all of the version between its first and last versions. Or, you can delete it altogether.

If you don't delete the branch, you can mark it as inactive, which is a message to other programmers that they probably don't want to add other versions to the branch.

You can designate a successor version, which will usually be the result of merging the branch into another branch.

Version Control Pathnames

A VC file image is identified by a pathname, a VC branch, and a VC version within the branch. You can reference a file image by using pathnames that include version control information. Ordinary pathnames have the form:

host:*directory*>*name*.*type*.*version*

A version control pathname has additional information at the end:

host:*directory*>*name*.*type*.*version*◊*vc-branch*.*vc-version*

Note the ◊ (SYMBOL-ESCAPE) character that delimits the version control information. Like the : that delimits hosts, it is the same for all types of pathnames.

A VC pathname includes both an ordinary version and a VC version. However, the correct contents of a file image are always in the *newest* ordinary version. Unless you are debugging VC, you should never use non-newest ordinary pathname versions in VC pathnames.

The VC information in pathnames interacts with merging and defaults just like the rest. You will occasionally find yourself with a default that includes VC information when you want to reference a non-VC file. To avoid merging in the VC information from the default, put a trailing ◊ on the end of the pathname you type.

SCT Integration

So far, we have described Version Control as organizing individual source files. Since we use SCT to organize most of our source, Version Control is integrated with SCT.

Target Machine Versus Source Branches

SCT, like **make-system** before it, can manage separate lines of system versions for each of a set of *target machines*. At the moment, the LM2 and 3600 are wired in. #+/-feature is used (both in the **defsystem** and in the code proper) to control target-specific features. Different machines are assumed to use binary files with different file types, so that they can all be stored in the same directory.

A person might ask, (and several people have,) "Why not just use a Version Control branch distinction to keep track of the code for different machines, instead of #+/- stuff?" The answer is this: For small differences in the code, it is far easier to read #+/- conditionalizations in line than to look at two different copies of the definition, no matter how cleverly presented. For large differences, there is no point to considering the two version to be parallel version control branches, since they will never be merged together. It makes more sense to just have completely different source files.

The result is that the target machine support in SCT remains, and is not linked to Version Control branch support except in as much as #+/- conditionals can be used in the **defsystem** form to conditionalize the syntax described below.

System Branches and File Branches

The version controlled files containing a system's source can contain arbitrary branches; for example: private branches for temporary saving of today's patch, private branches used by one person to develop a new feature over a period of time, and public branches that distinguish major group development efforts. When compiling a system, the programmer wants to be able to say something like:

```
Compile System Vice-President-Simulator :Branch Lang
```

and have SCT compile the newest version of an appropriate branch of each file. The definition of 'appropriate' is not obvious. Consider the following scenarios:

1. Group X needs to produce a specially modified version of system Y. They want to take the current stable code base, apply some modifications, and ship the result. They *don't* want to include any development work done by anyone else to the rest of the system.

   In Version Control terms, this group wants to establish a branch in *all* of the files of the system, and make their changes against that branch.

2. Group I needs to develop modifications to some parts of system S. They don't need to change the rest of the system, and they want to continue to pick up changes made there by the rest of the development group.

   In Version Control terms, this group wants to specify that if a file *has* a branch named I, then compile from there. However, if the file has no such branch, then the compilation should look in some other specified branch.

To handle both of these cases, it is not sufficient to just take the branch name typed by the user and look for a branch with that name in each of the files. Instead, *system branches* are defined as mappings to appropriate file branches.

To declare that a system is managed by version control, you have to give it the **:version-controlled t** attribute. This declares that a system branch has to be specified to access the source of the system, and in particular, to compile it.

By default, all source files and component systems of a **:version-controlled t** system are version controlled. You can use the **:version-controlled nil** attribute for a module to include non-version-controlled source in the system.

By default, when you specify a system branch to an SCT interface, SCT uses that branch as the file branch for each file. This is sufficient for simple cases. For more complex situations, you can add mapping rules to the **defsystem**. The mapping rules map a system branch name to an ordered set of file (or component system) branch names. When compiling, SCT will look for the listed file branches *in order*, and use the first one that it finds. (**Note**, mapping to more than one branch is not yet supported.)

The branch specifications in a system might look like this:

```
(defsystem color
    (:pretty-name "Color"
     :default-pathname "SYS:COLOR;VC;"
     :maintaining-sites :SCRC
     :patchable t
     :distribute-sources t
     :journal-directory "SYS:COLOR;"
     :version-controlled t
     :default-system-branch "Release-7"
     :branch-mapping (("Release-7-0" "Release-7-0"))
                      ("Release-7" "Release-7"))
  (:module sync-programs sync-programs (:type :system))
  (:module support color-support (:type :system))
  (:serial  support
   "simple-color-hardware" ;; color map and control stuff shared everywhere
   "common-color" ;; shared general color stuff
   "managed-map"  ;; color map management utilities
   "cad-buffer"          ;; cad buffer
               ;; "cad-buffer-ii"     ;; cad buffer II
   "frame-grabber";; frame grabber
   "std-color"  ;; standard color controller,
   "hires"      ;; hires and 10bit paddle card
   "chroma";; chroma paddle card
   "lores"      ;; ntscs paddle card
   "commands"
   )
  (:module documentation
   ("sys:color;doc;color.text")
   (:type :text))
  (:module sync ("sys:color;sync;cad-buffer-tek-sync"
                 "sys:color;sync;cad-buffer-amtron-sync")
          (:version-controlled nil)
          ;;these have to be preloaded for color console to boot
          (:type :lisp-load-only)))
```

System Branches and Component Systems

The mapping questions described above exist for component systems as they do for files. A group working on some major change to Statice would want to establish a branch of the Statice system, and then specify that for some of the components they wanted to share the ongoing development version with the rest of the group, while for others they wanted to establish their own branch.

You can do this by using the **:branch-mapping** option to a *module* to specify the mapping for a particular system.

Version Control and System Versions

So long as a system has only one branch, it will appear to have the same set of integer major version numbers that systems have today. In fact, for all branches other than the default branch, the branch name will be apparent in the file name, as in `System-IMach-2-2.lisp`. At SCRC, we would rarely use the default branch, so that we would be typically dealing with files like `System-Rel7-0-349-100.lisp`.

Note: SCT does not quite support compiling from more than one branch.

Version Control and SCT Journals

Journal directories will be per-branch.

Modification Comments

A *modification comment* is a human-readable record of a change to a version controlled file. Modification comments are created by users editing version control files. They are available to other users as a record of the change.

See the section "Showing, Hiding, and Adding Modification Comments".

Version Control Patch Files

Version Control includes facilities for patch files. Unlike existing patch files, which contain copies of the text of each item patches, version control patch files contain *references* to version control files. Compiling the patch file reads the text of each section out of the appropriate version of the version controlled file.

A version control patch source is a file of canonical type **:vc-patch** (surface type "vcp"). It can contain references to patch sections in any language. It compiles, with the Compile File CP command, into an ordinary patch .BIN file.

Eventually, version control patches will be used for ordinary system patches. For now, they are only used to make private patches from branches of files or systems.

See the section "Using VC Patch Files for Parallel Development".

Using Version Control

Version Control Cookbooks

This part of the document provides scripts for doing typical operations. Currently, most interfaces to the version control regime are implemented through Zmacs, but some are CP commands.

You begin either by converting an existing non-version-controlled file into a version-controlled file, by creating a new version-controlled file, or by reading in a version-controlled source file.

Converting Source to Version Control

There are two commands that take groups of flat files and convert them to a single version control file.

- **Convert File Sets to VC Files** This command takes a series of pathnames with version fields of **:wild** (`*`), for example, `sap:>Margulies>lispm-init.lisp.*`, and a target directory pathname, and makes a VC file for each set of flat files. See the section "Convert File Sets to VC Files CP Command".

- **Convert System Sources to VC Files** This command takes an SCT system, and converts the flat source files into VC files. See the section "Convert System Sources to VC Files CP Command".

To convert a system to version control, do the following:

- Use the `Convert System Sources to VC Files` command to convert all of the Lisp source files.

- Delete the VC file for the sysdcl file. The command is too stupid to skip it, even though SCT cannot use VC sysdcl files.

- Modify the **defsystem** as follows:

  ° Change the default pathname (and any other source pathnames) to the pathname of the VC files. **This will *not* be a VC pathname.**

  ° Add a **:version-controlled t**, **:default-system-branch** and a **:branch-mapping** option. See `VERSION-CONTROL:EDITOR-SUPPORT;SYSDCL.LISP` for an example.

  ° If you have SAGE sources or other source that you don't want to put into version control, put them in their owm module or module-group with a **:version-controlled nil**.

  ° Set the output pathname to be where it used to be, instead of the VC subdirectory.

  ° Set the journal file pathname to where it used to be, instead of the VC subdirectory.

- Recompile the system:

```
(compile-system :foo :system-branch "Initial")
```

Create a New Version-controlled Source File

If you are starting from scratch, the Create VC File Buffer (m-X) Zmacs command creates a buffer with an attribute line and a single, empty program section. This buffer is associated with a pathname into which it will be written.

The Split Hard Section (m-x) (bound to s-o) command is then used to create additional empty sections.

Read in an Existing Version-controlled Source File

If you supply a full VC pathname, ZMACS will read the specified file image into a buffer.

If you don't supply a full VC pathname, Find File (c-X c-F) detects that a file specified is a version-controlled file, and queries you for the VC branch and VC version. The available branches are listed in the typeout window. Mouse clicks are available to show the versions in a branch or to take the latest version of a branch.

Modify a buffer

When a VC buffer is first created, it is "read-only" so any number of users can view the same file version. Version Control ensures that users don't collide in trying to create new versions. So you can't start modifying your buffer without indicating if and how you intend to save it, so that other users can be prevented from colliding.

When you try to modify a read-only VC buffer, or use the Set Buffer Disposition (m-x) command, the editor ask you how you plan to dispose of your changes, with the query: "Modification Style?" Depending on the context you are in, you are presented with some or all of four choices: "P, B, D, or N."

- P (Private Branch) -- Create a new private branch and set this buffer to be saved as the first version in it. Branches that are private to a person are normally not seen by other programmers interacting with a version controlled file. You might want to create a private branch if you are experimenting and you do not want your experiments to interfere with other changes made to the code by others. ⊗ A Zmacs command exists to show private branches associated with a file.

- B (Public Branch) -- Create a new public branch and set this buffer to be saved as the first version in it. Public branches are seen by all users of the version controlled file.

- D (Disconnect Buffer) -- Make this buffer modifiable without preparing it to be saved. You have to specify its disposition before saving it.

- N (Next in Branch) -- Lock the branch for this buffer and insure that the buffer contains the latest version in the branch, reading in a newer version if necessary. Then set this buffer to be saved as the next version in the branch. See the section "Merging a Branch into Another Branch".

This query is here to save you from having to sort out the results of two people modifying the same source at the same time and then having to merge their changes later. If you like to live dangerously, you can always make disconnected buffers, and hope that no one else modifies the same file branch before you get around to trying to save.

For example, you disconnect a buffer, modify it, and then use Set Buffer Disposition (m-X) to specify Next in Branch. If someone else has added a new version to this branch since this buffer was disconnected, then you will have to merge those changes with your changes.

You can set variables in your lispm-init to get automatic choices under some circumstances. See the section "Customizing Your Interaction with Version Control".

Dealing with Problems with VC Locks

Network problems and software bugs can cause inconsistencies in file and branch locks. In particular, the file server can hold a lock on behalf of a host that no longer thinks that it holds the lock, or which has crashed.

branch lock problems

When you try to set up a buffer for "Next in Branch" modification and the branch is locked, the host holding the lock and the time it has held it are shown in the typeout window. If you suspect that the lock is invalid, you should try to verify this on the machine that claims to hold the lock, by looking for modified editor buffers or other version control activities in progress. If the machine has crashed since the time the lock was acquired, then you can be sure that the lock is spurious. When you are convinced that the lock is spurious, proceed as follows:

- Recheck the status with Show Branch Lock Status (m-X).
- If the lock is still held by the same machine, use Break Branch Lock (m-X) to break it.
- Proceed to modify the buffer.

See the section "Show Branch Lock Status ZMACS Command". See the section "Break Branch Lock ZMACS Command".

file lock problems While branch locks are help for a long time (as long as you have a buffer modifiable), file locks are only help for long enough to read in the current version, modify the file, and write it back out again. Therefore, problems with file locks are much less frequent.

If your machine trys to lock the file lock and fails to lock it, you will enter the debugger. Unless you have positive evidence

that the lock is invalid, such as a crashed machine, you should use one of the proceed options that continues to wait for the lock to be unlocked. Breaking a file lock that another machine holds legitimately can lose information, while breaking a branch lock will just force another user to merge their changes.

Reverting and Unmodifying VC a Buffer

m-X Revert Buffer works for version control buffers. If the buffer has been modified, or even made modifiable, you are queried as to whether you want a read-only copy or a clean copy with the same modification style set as you had before.

m-~ is equivalent to m-X Revert Buffer, since the only way to make the buffer "unmodified" is to restore its original contents.

Make a New Branch by Making a Branchpoint

Version-controlled buffers are initially in a read-only state. When you attempt to modify a file, you must answer the "Modification Style?" query. To make a branchpoint, you answer either "P" (create a private branch) or "B" (create a public branch). Alternatively, you can issue the command Set Buffer Disposition (m-X), which invokes the same "Modification Style?" query.

Creating, Killing, and Moving Hard Sections

Split Hard Section (m-X) uses the cursor position to mark off an area as a new hard section. If the cursor is at the end of the line, an empty hard section is created following the line. If the cursor is at the beginning of the line or anywhere on the line, the new hard section is created before the line the cursor is on. Split Hard Section is bound to s-o.

Kill Hard Section (m-X) marks a hard section deleted and and pushes it on the kill ring. It remains in the buffer with a "Deleted" indication in its title diagram until you save the buffer, or yank it someplace else. If you yank a killed hard section into buffer from which it came, it regains its old identity, and the "Deleted" copy is removed. If it is yanked into a different buffer, then it becomes a new hard section. Note that killing a hard section is different from killing all of its contents. If the user just kills the contents, then the empty hard section remains in the buffer. If you don't want to yank it, be sure to pop it off of the kill ring.

You can restore a killed hard section with UnKill Hard Section (m-X). Both Kill and Unkill are available as mouse gestures on the section title.

To move a hard section, first kill it with Kill Hard Section (m-X), move the cursor to the place you want it, and then yank (c-Y) the hard section.

Controlling the Names of Hard Sections

Hard sections have no names. However, the editor scans their text to come up with a useful title. The editor will use the name based on the function spec and definition type of the first definition of the hard section as the title of the hard section.

If you have a hard section with no definition at all, or a bunch of definitions that you want to group under some descriptive title, use the macro **zwei:define-section-name**.

zwei:define-section-name *name-symbol* &rest *ignore* *Macro*

This macro establishes a named definition for the purposes of hard section title. The first argument names the section, and the rest are ignored. This macro expands into nothing, so that it has no effect on evaluation or compilation.

Showing, Hiding, and Adding Modification Comments

The Zmacs command Show Modification Comments (m-X) makes modification comment regions appear in the buffer. For each hard section, a modification comment region appears after the hard section. The modification comment region in turn contains headings for each definition in the hard section.

You add modification comments by using normal editor commands in the modification comment regions.

Since modification comments can clutter up the editing screen, you can can hide modification comments with the Hide Modification Comments (m-X) Zmacs command.

Saving

There are two kinds of saving: *normal saving* and *emergency saving*.

- The normal case: The uses uses ordinary zmacs commands such as c-X c-S or m-X Save File Buffers. The buffer is written out as a new version of the version-controlled file. The encached file is written to disk at this time. Modified encached files are never retained in virtual memory. At save time, the editor decides on the connection between hard section currently in the editor buffer and the hard sections in the version of the file that you read in. The editor considers two (possibly conflicting) data:

  - For each hard section read from the file, the text in it at save time presumably is the next version of that section.

  - For each definition in the buffer, the copy of the definition at save time is presumably the next version of the section that contained that definition in the version read from the file.

To simplify the problem, the editor only considers the *first* definition in each section when establishing these historical links. Note that each section read in from the file has an internal section-id number which it retains until the save regardless of the content of the section.

The editor resolves these data as follows:

1. Any section which contains the same first definition as it did when it was read retains its section-id in the new version.

2. Any section that fails the first test, but contains the same first definition as some other section that failed the first test, takes the seciton-id of that section.

3. Any sections that fail the first two tests are treated as follows: for each one, if the first two passes didn't associate some other section with its section-id, it keeps its section-id. Otherwise it is treated as a new section.

If a section has no definitions, then it can't participate in the first two passes, unless you explicitly declare a name for it. See the section "Controlling the Names of Hard Sections".

- ⊗ The emergency case: In an emergency (probably indicated by the Save File Buffers command) the user wishes to write out the buffer in a hurry. Since writing a new version into a version-controlled file requires the writer to hold a lock on the file, and since the lock may be held by someone else for an extended period, it is not possible to write out a version into a version-controlled file. Instead, an image of the buffer is written into a special file. Later, the user can read in the special file, merge as necessary with changes made to the encached file, and then write out the changes as a new version-controlled file version.

In either case, the user is given an opportunity to provide modification comments for any modified hard sections that do not yet have comments.

Merging a Branch into Another Branch

A Zmacs command compares the terminal version of the two branches, using the version before they branched as a reference. The first time you merge two branches, the reference version is their common ancestor. The next time you merge them, the reference version is the version you produced by merging the last time.

If there are no hard sections changed differently in both branches, the merge proceeds without intervention. When both versions modified the same hard section, branch merging sets up annotations that describe the different versions, and you have to choose the version you want yourself.

One of the two branches is identified as the *source*, and the other is identified as the *target*. Changes made to the source are merged into the target, creating a new

version of the target. The source remains unchanged. In one scenario, the source is the installed branch of a program while the target is the branch under development. Merging allows the development branch to pick up maintenance changes that have been made to the installed branch. In a second scenario, a development project has been completed: the source is the development branch, while the target is the installed branch. Merging installs the results of development, being careful about any other changes that may have been made to the installed branch in the meantime. For additional information,See the section "Using VC Patch Files for Parallel Development". The command Merge VC File Branches (m-X) runs branch merging on a single file, while Merge VC System Branches (m-X) runs branch merging on the files of an entire system. See the section "Merge VC File Branches ZMACS Command".See the section "Merge VC System Branches ZMACS Command". After merging, you use one of the following commands to resolve conflicts between the changes in the two branches, using annotations created by the Merge VC File/System Branches command. Edit Compare Differences (m-X) processes a single file, while Tags Edit Compare Differences (m-X) processes the files designated by a Zmacs tags table, such as the tags table automatically set up by Merge VC System Branches (m-X). See the section "Edit Compare Differences ZMACS Command".See the section "Tags Edit Compare Differences ZMACS Command".

Using VC Patch Files for Parallel Development

If you have a source change stored as a branch to a file or several files of a system, you can use version control tools to make a patch that includes your changes. This is much more convenient than maintaining a conventional private patch file. There are two ways to do this: first we describe how to make a VC-specific private patch file, then we explain how to use Version Control aid in constructing patch files of the normal SCT variety.

VC-specific Private Patch Files

At this time, there are some restrictions. All of the changed hard sections in a file will be put in the branch in the order that they occur in the file. If you need a different order, you can edit the .vcp file to rearrange the references. For a system, hard sections for files are ordered according to the SCT dependencies. There are no special provisions for patch only code. To include additional code in the patch, you have to put it in ordinary hard sections which you can delete before merging your branch back into the main line of development. To keep code out of the patch altogether, you have to edit the .vcp file. These restrictions will be removed in future versions.

There are two CP commands that make VC patch files:

Make VC Private Patch File

Use Make VC Private Patch File if you know that all of your changes are in a single file. It takes as arguments the file pathname (this is *not* a VC pathname), the branch name, and the output file pathname, and creates a .vcp file. You can com-

pile the file with the Compile File CP command. As mentioned above, you can edit this file, which contains a series of lisp forms that specify the references to hard sections. See the section "Make VC Private Patch File CP Command".

Make VC System Branch Patch File
: Use Make VC System Branch Patch File if your changes are spread across several files of a system. It takes a system, a branch, and an output file. The branch need not be one of the branches defined in the **defsystem** for the system. Generally, only branches intended for compiling the entire system are recorded there (and compiling multiple branches isn't quite supported yet). However, `<help>` can only help you with branches that are listed in the **defsystem**. As with Make VC Private Patch File, the result of the command is a `.vcp` file. See the section "Make VC System Branch Patch File CP Command".

VC patch files reference specific versions of hard sections. So if you make further changes in your branch, you have to remake the patch file to pick up the newest information.

Constructing Patch Files of the Normal SCT Variety

You can start a patch using the Zmacs Add Patch (m-X) command, and then issue a CP command to add sections to that patch either from a single VC file or from all of the VC files in a system. You then go back to Zmacs to finish the patch, if necessary editing it first. For example, you might add patch-only forms to fix up the world to the patch.

The sections to be patched are those that differ textually between two versions of the file or files. These can be two versions of two different branches, or two versions of a single branch.

See the section "Add Patch VC File Differences CP Command". See the section "Add Patch VC System Differences CP Command".

Customizing Your Interaction with Version Control

zwei:*query-on-vc-buffer-modifications* *Variable*

This variable controls how ZMACS queries you when you try to modify a version control buffer that is read-only. The default is to always query. The variable must contain a plist from keywords to values. Each keyword names a type of file version. If the value is **t**, then you are queried with the standard "What modification style?" query. If it is **nil**, then ZMACS determines the modification style from **zwei:*how-to-modify-a-vc-buffer***. The keywords are as follows:

:private-modify If **t** , queries on modifications of private buffers

:private-reconnect If **t** , queries on reconnections to private buffers

:public-modify If **t** , queries on modifications of public buffers

:public-reconnect If **t** , queries on reconnections to public buffers

:modify-non-leaf If **t** , queries on modifications of non-leaf versions, regardless of whether they are private or public.

:reconnect-non-leaf If **t** , queries on reconnections to non-leaf versions, regardless of whether they are public or private

Here is an example form for setting **zwei:*query-on-vc-buffer-modifications*.**

```
;;; In this example, all the keywords are set to t
(setq zwei:*query-on-vc-buffer-modifications*
      '(
        :private-modify t
        :private-reconnect t
        :public-modify t
        :public-reconnect t
        :modify-non-leaf t
        :reconnect-non-leaf t))
```

zwei:*how-to-modify-a-vc-buffer* *Variable*

This variable controls what action ZMACS takes when you modify a vc file buffer and **zwei:*query-on-vc-buffer-modifications*** suppressed the "What modification style?" query. Like **zwei:*query-on-vc-buffer-modifications***, this variable must contain a plist

In this case, however, the keywords can take any of five values.

:next-in-branch Locks the containing branch and prepares to make the next version

:private-branch Creates a new, temporary branch

:disconnect-buffer Renames the buffer to be unique so that you can read in a fresh copy of the file it came from without a name conflict. No lock is obtained for the disconnected buffer. This parameter is invalid for the **version-control-internals::reconnect** cases.

:ask Asks the user to select a modification style

:error Make the user's console beep. This is valid only for **version-control-internals::modify** cases. It is most useful for **:modify-non-leaf.** This overrides **zwei:*query-on-vc-buffer-modifications*.**

```
;;; In this example, all the keywords are set to :ask
(setq zwei:*how-to-modify-a-vc-buffer*
      '(
        :private-modify :ask
        :private-reconnect :ask
        :public-modify :ask
        :public-reconnect :ask
        :modify-non-leaf :ask
        :reconnect-non-leaf :ask))
```

Next is an example of a typical interaction style for an experienced user. When the user attempts to modify a buffer, a disconnected buffer is created automatically. When the user tries to save the buffer, they get the standard query.

```
(setq zwei:*query-on-vc-buffer-modifications*
      '(
        :private-modify nil
        :private-reconnect nil
        :public-modify nil
        :public-reconnect nil
        :modify-non-leaf nil
        :reconnect-non-leaf nil))

(setq zwei:*how-to-modify-a-vc-buffer*
      '(
        :private-modify :disconnect
        :private-reconnect :ask
        :public-modify :disconnect
        :public-reconnect :ask
        :modify-non-leaf :disconnect
        :reconnect-non-leaf :ask))
```

No automatic way to create a new public branch or merge exists. This has to be done explicitly by the programmer.

zwei:*vc-title-diagram-style* *Variable*

This variable controls the visual presentation of the section titles in version control buffers. Its value must be a keyword symbol chosen from the following list:

:filled-box The title is a reverse-video box containing the name of the first definition of the section. **zwei:*vc-filled-box-diagram-character-style*** controls the character style of the text in the box.

:text-and-rule The title is the name of the first definition of the section, with a narrow horizontal rule to separate it from the text of the section. **zwei:*vc-text-and-rule-diagram-character-style*** controls the character style of the title text.

zwei:*vc-center-title-diagrams* *Variable*

This variable controls whether section titles are centered on the ZMACS window. If it is **t**, they are centered. If it is **nil**, they are left-justified.

List of Version Control Commands

Zmacs Commands for Version Control

Create VC File Buffer Zmacs Command

Create VC File Buffer (m-X)

Creates a buffer for a new version controlled file. The new buffer has two empty sections, one for the attributes and one for the first definition.

By default, the buffer is initialized to create the first version of the branch named by **version-control-internals::*default-initial-branch-name***, which by default is "Initial". With a numeric argument, you are prompted for the name of the initial branch.

The new file is not created on disk until you save the buffer.

Show Modification Comments ZMACS Command

Show Modification Comments (m-X)

Makes the modification comment sections visible in the current buffer.

When you modify a VC file, you can leave out-of-band comments for any of the hard sections describing your changes. You enter these comments into the modification comment sections. By default these are not visible on the screen. To add modification comments, use this command to make the sections visible.

To see modification comments from previous changes, use Show Section History (m-X).

Hide Modification Comments ZMACS Command

Hide Modification Comments (m-X)

Removes modification comment sections from display in the current buffer.

For more information on modification comments, See the section "Show Modification Comments ZMACS Command".

Show Titles ZMACS Command

Show Titles (m-X)

Shows the full name of the first definition in a box at the top of each hard section. If **zwei:*zmacs-display-bubble-diagram-lines-default*** is **t**, then titles are shown by default. The title box is mouse sensitive for operations on the entire hard section.

See the variable **zwei:*zmacs-display-bubble-diagram-lines-default***.

Hide Titles ZMACS Command

Hide Titles (m-X)

Shows a thin horizontal bar at the top of each hard section as a mouse target for operations on the hard section. If **zwei:*zmacs-display-bubble-diagram-lines-default*** is **nil**, then this is the default for new buffers.

See the section "Show Titles ZMACS Command".

See the variable **zwei:*zmacs-display-bubble-diagram-lines-default***.

zwei:*zmacs-display-bubble-diagram-lines-default* *Variable*

Controls whether version controlled buffers have full hard section titles by default. If this variable is **t**, the default, then the name of the first definition in each hard section is displayed at the top of the hard section with mouse sensitivity for operations on the hard section.

If this variable is **nil**, then the default is to use a thin horizontal bar as the mouse target.

See the section "Show Titles ZMACS Command".

See the section "Hide Titles ZMACS Command".

Split Hard Section ZMACS Command

Split Hard Section (m-X) (bound to s-o)

Splits one hard section into two. This is the way you create a new hard section. To add a new hard section after an existing section, position the cursor on the diagram line at the bottom of the section and use this command.

Kill Hard Section ZMACS Command

Kill Hard Section (m-X)

Marks a hard section for deletion when the buffer is saved, and pushes it on the kill ring. If you yank it back (with c-Y) into a version controlled buffer, it will appear as a distinct hard section as opposed to being added to the section containing the point. If you yank it back in the same buffer that you killed it from, the killed copy will disappear in favor of the new copy.

UnKill Hard Section Zmacs Command

UnKill Hard Section (m-X)

Restores a hard section that has been marked for deletion. This is the opposite of Kill Hard Section (m-X). Use it when you change your mind about killing a hard section.

Show Section History ZMACS Command

Show Section History (m-X)

Creates and selects a buffer containing all of the versions of the hard section containing the point. This *history buffer* has one hard section for each version. The changes from one version to the next are highlighted in boldface.

Merge VC File Branches Zmacs Command

Merge VC File Branches (m-x)

Merges changes from one VC file branch into another. Prompts you in the minibuffer for a VC file pathname and two branches, the *source* and *target*. It compare-merges the newest versions of the two branches with respect to their common ancestor, and creates a buffer with the merged result. The new buffer is modifiable, and set up to be saved as next in the target branch.

See the section "Edit Compare Differences ZMACS Command".

Edit Compare Differences ZMACS Command

Edit Compare Differences (m-X)

Steps through annotations in the current buffer that were left by compare-merge to indicate places where automatic merging could not resolve a difference and manual intervention is required. At each difference, you can press a single key to resolve the difference and Edit Compare Differences will go on to the next difference. Alternatively, you can escape to the editor, examine and possibly modify the two changes, use editor commands explained below to resolve the difference, and then press c-. to go on to the next difference.

A typical difference looks like:

;;;;;COMPARE-MERGE Begin Difference
;;;;;COMPARE-MERGE Text in A - *pathname*
text for Ancestor
;;;;;COMPARE-MERGE Text in S - *pathname*
text for Source
;;;;;COMPARE-MERGE Text in T - *pathname*
text for Target
;;;;;COMPARE-MERGE End Difference

Note that each of the Text lines is of the form "Text for TAG". The tag identifies where the text came from: **S** for the source, **T** for the target, or **A** for their common ancestor.

If you press one of those letters, the associated piece of text will be inserted, the other pieces of text will be deleted, and Edit Compare Differences goes on to the next difference.

If you press Space or Rubout, Edit Compare Differences exits to the editor, leaving the annotation in place. If you press anything else, Edit Compare Differences exits to the editor and then executes the key you pressed as an editor command.

Once in the editor, you can use the following commands to resolve the difference. After resolving the difference, press c-. to go on to the next difference. As you resolve the difference, the text that you intend to be the result is inserted after the Begin Difference line.

`super-Z TAG`	Takes the text for tag `TAG` (usually S, T, or A) and inserts it after the Begin Difference.
`super-Z super-TAG`	Takes the text for tag `TAG` (usually S, T, or A) and uses it as the resolution of the difference, removing all of the annotation. This is a combination of `super-Z TAG` with `super-control-S`.
`super-control-S`	Removes the difference annotation comments, leaving the text (if any) that you have chosen as the resolution of the difference. While you are still positioned at this difference, this command will swap back and forth between the annotation and the currently chosen text.
`meta-Right`	Clicking meta-Right on one of the annotation lines selects the associated text, and inserts it after the Begin Difference.

Tags Edit Compare Differences ZMACS Command

Tags Edit Compare Differences `(m-X)`

Steps through all the compare-merge differences in a tag table. This command is just like Edit Compare Differences `(m-X)` except that it processes all of the files of a tag table instead of just one buffer.

See the section "Edit Compare Differences ZMACS Command".

Merge VC System Branches Zmacs Command

Merge VC System Branches `(m-X)`

Merges VC file branches across an entire SCT system. Prompts in the mini-buffer for an SCT system and two branches, the *source* and *target*. For each version controlled file in the system, do the equivalent of a Merge VC File Branches `(m-X)`. See the section "Merge VC File Branches ZMACS Command". That is, it

compare-merges the newest version of the source branch against the newest version of the target branch, and puts the results in a buffer set to be saved as the newest version of the target branch.

After all of the merging, the command collects all of the resulting buffers into a tag table, facilitating the use of Tags Edit Compare Differences (m-X). See the section "Tags Edit Compare Differences ZMACS Command".

Set Buffer Disposition ZMACS Command

Set Buffer Disposition (m-X)

Specifies how a VC buffer is to be saved. Prompts in the minibuffer just as trying to modify a read-only VC buffer prompts. Use this to change your mind and disconnect a buffer set to save as Next in Branch or vica versa. See the section "Modify a buffer".

List Modified Sections ZMACS Command

List Modified Sections (m-X)

Lists the names of all modified hard sections of the current buffer in the typeout window. c-. edits them one after the other. They are mouse sensitive.

Show Section Changes ZMACS Command

Show Section Changes (m-X)

Shows changes between different version of the current hard section in the typeout window. By default, it compares the version on display with the previous version. If the version on display has been modified since it was read in to the buffer, the comparison is between the modified copy and the version read in. If it has not been read in, the comparison is between the version on display and its parent version. With a numeric argument, prompts for two versions to compare.

Show All Section Changes ZMACS Command

Show All Section Changes (m-X)

Shows the changes from one version to another of all hard sections of the current buffer. In effect, this command is the same as Show Section Changes (m-X) for each hard section that has been changes from the previous version of the file. By default, the comparison is between the version on display and the previous version. With a numeric argument, the command prompts for explicit versions. See the section "Show Section Changes ZMACS Command".

Show VC File Branches Zmacs Command

Show VC File Branches (m-X)

Shows all the branches defined in the file of the current buffer. By default, it only shows public branches. With a numeric argument it shows private branches as well.

Goto Hard Section Beginning Zmacs Command

s-< *or* Goto Hard Section Beginning (m-X)

Moves the point to the beginning of the current hard section. If the point is already at the beginning of the hard section, it moves to the beginning of the previous hard section.

Goto Hard Section End Zmacs Command

s-> *or* Goto Hard Section End (m-X)

Moves the point to the end of the current hard section. If the point is already at the end of the hard section, it moves to the end of the next hard section.

Find File Ignoring Version Control ZMACS Command

Find File Ignoring Version Control ZMACS Command (m-X)

Reads a file into the editor as a flat file, ignoring any version control information. This commands is only used when debugging version control. It allows you to edit the internal stored representation of a VC file.

Show Branch Lock Status ZMACS Command

Show Branch Lock Status (m-X)

Queries the lock server (generally the file server) for the locking status of a branch for the file in the current buffer. It prompts you for the branch name, defaulting to the branch that the current buffer was read from. The information is displayed in the typeout window.

See the section "Dealing with Problems with VC Locks".

Break Branch Lock ZMACS Command

This command forcibly unlocks a branch of the file in the current buffer, even though the lock server records a lock for some host. It prompts you for the branch name, defaulting to the branch that the current buffer was read from.

Use the command with care! See the section "Dealing with Problems with VC Locks".

Command Processor Commands for Version Control

Convert File Sets to VC Files CP Command

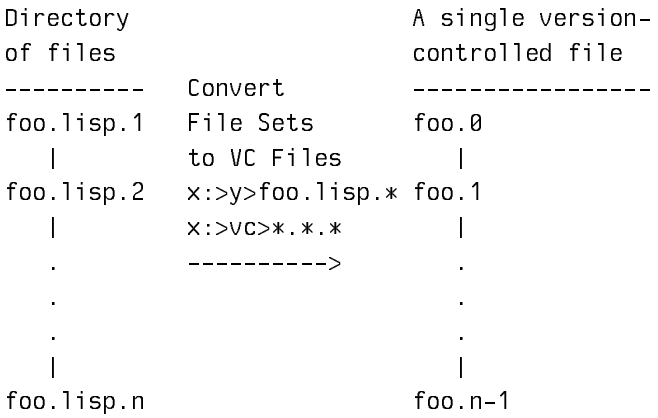
Here is the meta-Complete menu for this command:

```
Command: Convert File Sets to VC Files
File set: SAP:>MMcM>greenhouse>graphics>postscript>s.lisp.*
Target directory: SAP:>MMcM>greenhouse>graphics>postscript>s.lisp
Verify: Yes No
Branch-Name: "Initial"
New-Branch-Root: the parent version for a new branch, or nothing
Create-Directories: Yes No
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

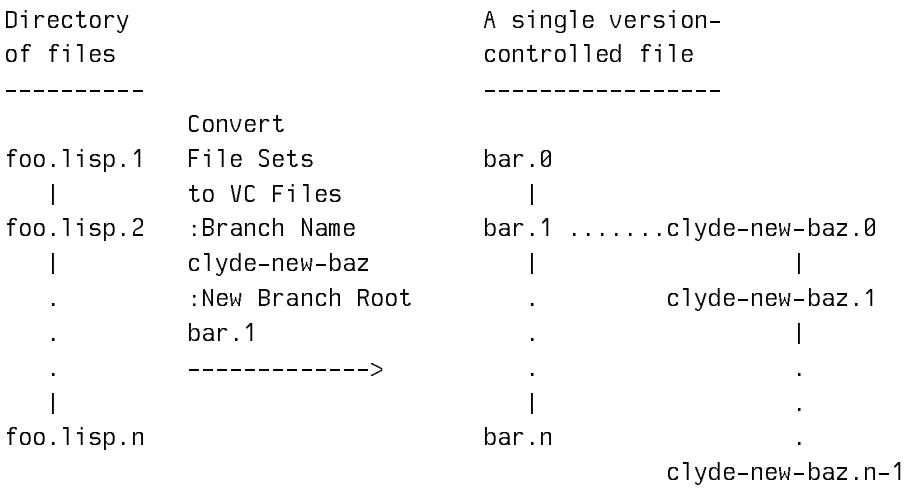
This command takes a sequence of file set pathnames and a target directory pathname. Each file set pathname is a pathname with a wild version. (Other components can be wild as well). For each distinct set of files (of the form sap:>Margulies>foo.bar.*), the command looks for a existing VC file with that name and type in the target directory. If none exists, a new one is created, and all of the available flat versions are represented as successive versions of the branch.

Each set of files *foo.lisp.1, foo.lisp.2,...foo.lisp.n* is converted into single version-controlled file with multiple versions inside it.

If a VC file exists, then all of the flat versions that aren't already in the file are added as additional versions along the branch.



If the target directory exists but the branch name you specify is new, then you must specify a value for the **:New Branch Root** keyword. The files are appended to the branch specified by the **:New Branch Root** keyword.



If the target file doesn't yet exist and you specify a branch name with the keyword **:Branch Name**, then the new file is created with the specified branch name only. This overrides the default case, which is to start with a branch name called Initial.

The keywords **:Branch Name** and **:New Branch Root** also make it possible to convert two existing sets of files into a single file containing multiple branches. To do this, pick up the main line with Convert File Sets to VC Files as usual. Then use the **:Branch Name** keyword to give the second line a branch name. This branch is attached to the main branch at the version specified after the **:New Branch Root** keyword. Here is an example.

```
Convert File Sets to VC Files
        "q:>moon>greenhouse>flavors>defgeneric.lisp"
        "q:>moon>trash>vc-tests>"
        :Branch Name other
        :New Branch Root initial.3
```

This command line takes all versions in greenhouse>flavors and appends them to trash>vc-tests>defgeneric.lisp in a branch named **other,** where the common ancestor is version 3 of branch **initial.**

If the value of the **:Verify** keyword is "Yes," then the files are re-read and compared to the stored version in the version-controlled file. By default, version control always verifies such conversions.

See the section "Converting Source to Version Control".

Convert File Tree to VC Files CP Command

This command takes files representing parallel developement in flat files and converts them to version control files. Parallel development is represented in flat files by making a snapshot of the file(s) as of some point in a new directory.

Convert File Sets to VC Files can also convert multiple sets of files in multiple directories, but you have to run the command once per branch. This command has the additional feature that you can construct a template that specifies where all the directories are and what files should be taken from each. Because this template gets complicated in a hurry, you write it into a file rather than typing it as a command argument.

The template is a list form. The top-level syntax is:

```
(:branch "branch-name" nil spec1 ... specN)
```

Many of the specs involve pathnames. The names and types in these pathnames are not relevant. They just specify the directory and the version.

branch-name is the name of the top-level branch to be created.

Each *spec* specifies a file to represent in the VC file. The specs can be:

a newest or oldest pathname
: takes the newest or oldest file from the directory.

(:after *after-pathname pathname*)
: *after-pathname* is either a newest or oldest pathname. The first file after (version-number-wise) the *after-pathname* in the directory specified by *pathname* is specified.

(:range *from to*)
: *from* can be a pathname or a :after spec. To must be a pathname. All the file version from *from* to *to*, inclusive, are included in the branch.

(:branch *name flag spec1 ... specN*)
: this specifies a new branch. *name* is the name of the branch. If *flag* is **t**, then a copy of first version in this branch is included in its parent at this point. If it is nil, then no copy is inserted, and the parent version of the first version of the branch will be the previous file specified.

Here is an example:

```
(:branch "Development" nil
         (:branch "Release-6-1" t
                  #p"q:>rel-6>sys>zmail>↔.↔.oldest"
                  #p"q:>rel-6>sys>zmail>↔.↔.newest")
         (:branch "Release-7-1" t
                  #p"q:>rel-7-0>sys>zmail>↔.↔.oldest"
                  #p"q:>rel-7-0>sys>zmail>↔.↔.newest")
         (:range (:after #p"q:>rel-7-0>sys>zmail>↔.↔.oldest"
                         #p"q:>rel-7>sys>zmail>")
          #p"q:>rel-7>sys>zmail>↔.↔.newest"))
```

This creates a branch named Development. The first version is copied from the first version of a branch named Release-6-1, which gets the oldest and newest files from the `>rel-6>` directory. The next version is copied from the first version of a branch named Release-7-1, which gets the oldest and newest files from the `>rel-7-0>` directory. Then, *all* files from the `>rel-7>` directory newer than the oldest file in the `>rel-7-0>` directory are taken.

Here is a `m-complete` menu for the command:

```
Command: Convert File Tree to VC Files
Tree spec file: the pathname of a file
Files to convert: the pathnames of one or more files
Template for target pathnames: the pathname of a file
Create-Directories: Yes  No
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

Convert System Sources Tree to VC Files CP Command

This command takes SCT system files representing parallel developement in flat files and converts them to version control files. Parallel development is represented in flat files by making a snapshot of the file(s) as of some point in a new directory.

Convert File Sets to VC Files can also convert multiple sets of files in multiple directories, but you have to run the command once per branch. This command has the additional feature that you can construct a template that specifies where all the directories are and what files should be taken from each. Because this template gets complicated in a hurry, you write it into a file rather than typing it as a command argument.

The template is a list form. The top-level syntax is:

```
(:branch "branch-name" nil spec1 ... specN)
```

Many of the specs involve pathnames. The names and types in these pathnames are not relevant. They just specify the directory and the version.

branch-name is the name of the top-level branch to be created.

Each *spec* specifies a file to represent in the VC file. The specs can be:

a newest or oldest pathname
: takes the newest or oldest file from the directory.

(:after *after-pathname pathname*)
: *after-pathname* is either a newest or oldest pathname. The first file after (version-number-wise) the *after-pathname* in the directory specified by *pathname* is specified.

(:range *from to*)
: *from* can be a pathname or a :after spec. To must be a pathname. All the file version from *from* to *to*, inclusive, are included in the branch.

(:branch *name flag spec1 ... specN*)
: this specifies a new branch. *name* is the name of the branch. If *flag* is **t**, then a copy of first version in this branch is included in its parent at this point. If it is nil, then no copy is inserted, and the parent version of the first version of the branch will be the previous file specified.

Here is an example:

```
(:branch "Development" nil
         (:branch "Release-6-1" t
                  #p"q:>rel-6>sys>zmail>↔.↔.oldest"
                  #p"q:>rel-6>sys>zmail>↔.↔.newest")
         (:branch "Release-7-1" t
                  #p"q:>rel-7-0>sys>zmail>↔.↔.oldest"
                  #p"q:>rel-7-0>sys>zmail>↔.↔.newest")
         (:range (:after #p"q:>rel-7-0>sys>zmail>↔.↔.oldest"
                         #p"q:>rel-7>sys>zmail>")
          #p"q:>rel-7>sys>zmail>↔.↔.newest"))
```

This creates a branch named Development. The first version is copied from the first version of a branch named Release-6-1, which gets the oldest and newest files from the `>rel-6>` directory. The next version is copied from the first version of a branch named Release-7-1, which gets the oldest and newest files from the `>rel-7-0>` directory. Then, *all* files from the `>rel-7>` directory newer than the oldest file in the `>rel-7-0>` directory are taken.

Here is a `m-complete` menu for the command:

```
Command: Convert System Sources Tree to VC Files
Tree spec file: the pathname of a file
System to convert: a system
```

```
Include-Components: Yes  No
VC File Subdirectory: "VC"
Create-Directories: Yes  No
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

Convert System Sources to VC Files CP Command

This command has the same basic function as `Convert File Sets to VC Files`, but processes all the files of an SCT system at once. It dosen't have all of the flexibility of `Convert File Sets to VC Files`, though.

Here is the `meta-Complete` menu:

```
Command: Convert System Sources to VC Files
Enter a system: Postscript
Vc-File-Subdir: "VC"
Create-Directories: Yes  No
Verify: Yes  No
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

For each source file (of type .lisp) of the specified system, this command converts flat sources to a VC file in the specified subdirectory. For example, if a system source pathname is `SYS:IO;BAND.LISP`, and the subdirectory name is "VC", then the VC file will be `SYS:IO;VC;BAND.LISP`.

The **Verify** keyword is the same as for Convert File Sets to VC Files.

The **Create Directories** keyword will create the subdirectory if it does not exist.

See the section "Converting Source to Version Control".

Make VC Private Patch File CP Command

This command takes all of the hard sections that have been changed or created in a particular branch of a VC file and assembles a VC patch file that references them in the order that they occur in the newest version of the branch.

Here is the `meta-complete` menu for this command:

```
Command: Make VC Private Patch File
VC File pathname: the pathname of a file
Branch name: file branch name
Patch file pathname: the pathname of a file
Patch-Note: a string
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

See the section "Using VC Patch Files for Parallel Development".

Make VC System Branch Patch File CP Command

This command takes all of the hard sections that have been changed or created in a particular branch of all of the VC files of a partular system and assembles a VC patch file of them. The sections are ordered in the patch by file; the files are ordered by SCT compilation dependencies, and the sections within each file by their order that they occur in the newest version of the branch. This command ignores non-VC source files of the system.

```
Command: Make VC System Branch Patch File
Enter a system: a system
Enter a system branch: a system branch
Enter the pathname of a file: the pathname of a file
Include-Components: Yes  No
Patch-Note: a string
Output-Destination: a destination
<abort> aborts, <end> uses these values
```

See the section "Using VC Patch Files for Parallel Development".

Show VC File Versions CP Command

This command lists the branches and version defined in version controlled files. By default, it lists the branches, but each is mouse sensitive with an action of showing all the versions. With the :Detailed Yes option, it shows all of the versions of all of the branches. You can supply a sequence of wild pathnames.

```
Command: Show VC File Versions
VC file pathnames: one or more pathnames
Detailed: Yes  No
Output-Destination: a destination
<end> uses these values, <abort> aborts
```

Extract VC File Version CP Command

This command makes a flat file containing an image of a file version of a VC file. If you give it a full version control pathname, then it extracts the version specified by the pathname. If you just give it the flat pathname of the version controlled file, it prompts for an explicit file version.

Be sure to give a *different* pathname for the output file than for the version controlled file.

```
Command: Extract VC File Version
Enter the pathname of a file: SAP:>Margulies>lispm-init.lisp.newest
Enter a version control file version: a version control file version
Enter the pathname of a file: SAP:>Margulies>lispm-init.lisp.newest
Output-Destination: a destination
<end> uses these values, <abort> aborts
```

Fork VC System CP Command

This command makes a new branch in each of the version controlled source files of an SCT system. You specify a system major version. It takes each version controlled source file, and forks the version of that file that was compiled for the specified major version. The fork is made by copying the contents of the file version into a new file version which is the base of a new branch.

This command is used for source splits. It creates an independent branch in which changes are made.

```
Command: Fork VC System
Enter a system: Zmail Debugging Tools
Enter a version designator for Zmail Debugging Tools:
   a version designator for Zmail Debugging Tools
Enter a string: a string
Output-Destination: a destination
<end> uses these values, <abort> aborts
```

Add Patch VC File Differences CP Command

This command adds some sections to the current SCT patch. You must issue the m-X Start Patch command in Zmacs first, before using the Add Patch VC File Differences CP command. The sections to be patched are those that differ textually between two versions of the file. These can be two versions of two different branches, or two versions of a single branch.

The patch comments for the patch are initialized from the modification comments in the VC file. When finishing the patch, you may want to edit these comments to make them conform to patch-comment style.

Sections go into the patch file in the order that they appear in the source.

The arguments to Add Patch VC File Differences are

file The pathname of the VC file.

branch The name of the branch whose sections are to be patched. The newest version of this branch is always used.

The following are keyword options:

`:Base Branch` The name of the "reference branch". The sections that go into the patch are the ones that differ between this branch and the branch specified earlier. If no base branch is specified, the ancestor of the patching branch is used.

`:Base Version` The version of the "reference branch". The default if `:base-branch` is specified is newest, otherwise the default is the ancestor of the patchging branch. This argument can be newest, oldest, or an integer.

`:Patch Note` A patch note string, useful when creating a private patch.

Add Patch VC System Differences CP Command

This command adds some sections to the current SCT patch. You must issue the m-X Start Patch command in Zmacs first, before using the Add Patch VC System Differences CP command. The sections to be patched are those that differ textually between two versions of each file in a system. These can be two versions of two different branches, or two versions of a single branch.

The patch comments for the patch are initialized from the modification comments in the VC file. When finishing the patch, you may want to edit these comments to make them conform to patch-comment style.

Sections go into the patch file in the order that they would be seen by the compiler when SCT was compiling the system. This means that within a file, sections are patched in the order that they appear in the file, and between files sections are patched in the order determined by the dependencies in the **defsystem**.

The arguments to Add Patch VC System Differences are

system The name of the system that is the source of the patch. This should be the system that the current Zmacs patch is for, or a subsystem of it.

branch The name of the branch whose sections are to be patched. The newest version of this branch is always used.

The following are keyword options:

`:Include Components`
If Yes, component systems also serve as sources for the patch.

`:Base Branch`	The name of the "reference branch". The sections that go into the patch are the ones that differ between this branch and the branch specified earlier. If no base branch is specified, the ancestor of the patching branch is used.
`:Base Version`	The version of the "reference branch". The default if `:base-branch` is specified is newest, otherwise the default is the ancestor of the patchging branch. This argument can be newest, oldest, or an integer. Integers here tend not to be very useful, because version numbers in the various files of a system are usually not synchronized.
`:Patch Note`	A patch note string, useful when creating a private patch.