Symbolics Version Control Design and Implementation

Preface: Purpose and Audience of this Document

The Version Control system is a package of programs that provide source management for Symbolics' software development. A functional description of the system can be found in "Symbolics Internal Guide to Version Control," which is a prerequisite to understanding the implementation. This document can be read on-line by loading the system `version-control-doc`.

The audience of this document are the people working on the code of VC, either to maintain and extend it, to design its successor, or just to understand it.

The purpose of this discussion is to lay out the gross modularity of the system and to describe how its components work and fit together. It includes some information on the design-space, but this is by no means a complete *apologia* for all of the decisions that led to the current design. When discussing design alternatives, pride-of-place is given to those choices which are the most likely candidates to be revisited in the course of ongoing implementation.

Overview of the Version Control Implementation

Version Control is modularized into a collection of top-level patchable SCT systems. The large number of independent systems was chosen to facilitate replacement of large functional blocks. Some examples of this kind of replacement are: Statice instead of text files for the underlying data store, or a ZMACS successor instead of ZMACS as the editor interface. The current distribution of code amongst the systems will not necessarily permit such replacements seamlessly. However, as the design of such a replacement becomes clearer, the modularity can be honed. The goal is to minimize the extra work involved in, say, developing the Statice-based implementation while still maintaining and enhancing the file-based implementation.

Here is a list of the systems that make up the Version Control system:

vc-packages
: This contains all package declarations for all constituents. While it would be desirable to maintain the packages for a major part of the system along with the rest of the code for that part, the requirement that packages and their exports all be defined before any of their referenced can be compiled or loaded preempts that ambition.

vc-pathnames
: This contains the file access path that supports direct pathname access to VC file versions.

vc-file-substrate
: This contains the file storage model. This includes parsing VC files, creating and maintaining encached (in-virtual-memory) files, and writing encached files out to VC files.

lock-simple
: This contains a network lock server, which is used for concurrency control on access to VC files. It has its own documenta-

	tion, entitled "The Lock Simple Network Service," in `y:>version-control>design>lock-server-doc.sab`.
compare-merge	This contains substrate for an multi-way compare merge which includes the ability to use a "common original" version to resolve divergent versions. This is included in VC because many VC editor tools depend on it. Compare-merge, on the other hand, does not depend on the rest of VC.
vc-editor-support	This contains all the ZMACS interfaces to version control, including VC buffers and all of the other user interfaces available in the editor.
vc-interfaces	This contains high-level tools and interfaces, including the tools for converting flat files to VC files. Some of these tools either have interfaces in ZMACS, and perhaps those interfaces should be moved to vc-editor-support at some point.
version-control-doc	This contains the SAGE user documentation for version control as well as the present document.

A chapter of this document is devoted to each of these systems except `vc-packages` and `version-control-doc`. In addition, significant subtrate for version control is included in Genera proper. The most significant parts of this support are described along with the part of version control that is most relevant.

VC Pathnames Design and Implementation

The `vc-pathnames` system contains the file access path for version control pathnames. Most of the support for version control pathnames is part of Genera.

The term "version control pathname" is a misnomer: *all* Genera pathnames except dummies can store a VC file version in addition to the usual set of host, device, directory, name, type, and version. The VC file version is stored as two additional items: a branch name (**fs:pathname-vc-branch**) and a branch version (**fs:pathname-vc-version**). The term "VC pathname" refers to any pathname which has something other than **nil** in one of those slots.

VC pathnames exist for two reasons. First, study showed that any attempt to make SCT support the compilation of source stored in VC files would be intractable unless the VC file version could be stored in the pathname. The alternative would have been to change all of the code in SCT that passes pathnames around to pass something else that could reference a pathname and a VC file version.

Second, its desirable for ordinary programs to be able to get a source image out of a VC file without having their user interface changed to get a VC file version in addition to a pathname and to call special interfaces to get the information.

VC Pathnames Architectural Model

Version Control implements an abstract file system on top of any of our supported file protocols. All it requires is simple character files. It makes no use of length information, versions, or anything else. Independent of implementation considerations, the pathname: `sys:io;band.lisp◊IMach.12` means version 12 within branch IMach of the file `sys:io;band.lisp.newest`.

Opening such a pathname gives a stream that read that particular version. Probing it reveals whether such a version existed. Some other operations are less well defined, but then again, not all file system support all operations on ordinary files.

VC Pathnames Base Implementation

Since the Version Control model is build on top of the rest of the pathname model, it conceptually encapsulates the rest of the flavor **pathname**. Thus, it can be implemented as a series of whoppers on the pathname protocol.

The version control information in a pathname is stored in two new instance variables of the flavor **pathname**: **vc-branch** (a branch name) and **vc-version** (a branch version number, including things like **:newest**). These will be treated just like the existing pathname components, with important exceptions described below. In addition to **:oldest** and **:newest**, the VC version can be **:parent**. This specifies the version that is the parent version of the oldest version in the specified branch.

VC Pathnames Parsing and Printing

Parsing of VC information is simple. **fs:parse-pathname** looks for ◊ (Symbol Escape) and treats anything following it as a specification of the vc-branch and/or vc-version. The two are separated by the usual "." delimiter.

Printing is more complex. There are a wide variety of pathname messages which return strings for various uses. Many of them are implemented in terms of each other. For example, the default method for **:string-for-readable-printing** just turns around and send **:string-for-printing**. For this reason, a straightforward scheme of whoppers that append the `◊branch.version` won't work, because in some cases it will happen twice. Teaching the bottom level methods that actually construct strings to append the VC information would have meant sprinkling the VC support over all of the per-host pathname support, which would have made maintenance hard and the creation of new host support in the field even harder than it is already.

Instead, there are whoppers (actually whopper-substs, which are actually wrappers) for all of the **:string-for** methods. These whoppers use a special variable (**fs:*vc-pathname-string-append-in-progress***) to make sure that the information is only appended once.

String caching is another source of complexity. The **:string-for-printing**, is cached in pathname instance variables for efficiency. Since the **fs:*vc-pathname-string-append-in-progress*** can require the methods for **:string-for-printing** to return strings both with and without VC information, there are two caches: one with VC, and one without.

VC Pathname File Access Path

Genera includes **(FLAVOR:WHOPPER :FILE-ACCESS-PATH FS:ACTIVE-PATHNAME-HOST)**. This whopper signals an error on any attempt to get a file access path for a VC pathname.

The `vc-pathnames` system patches out that whopper and replaces it by a file access path for VC file versions. The file access path is an encapsulation: When an application operates on it, it can turn around and operate on an ordinary file access path to the host that stores the VC file.

The file access path is primarily concerned with **open**. It supports **:input** and **:probe** openings in element-type **character**. Other directions or element types result in an error signal.

If, at open time, the desired file has already been read in an encached, the file access path will return a stream that reads from the encached file. If not, it will return a stream that reads directly from the VC file, returning the text of a particular version.

To make some user interfaces work better, notably in the editor, the **:directory-list** operation is "supported" by stripping off the VC information and passing the result along.

There is a partial implementation of **:complete-string**. If the user has typed a pathname ending with an ◊, indicating that they want to include VC information present in the default, then the method obligingly appends the VC information and passes the result along to the underlying file access path. Otherwise, the method strips off the VC information from the default (or what the user typed) and passes the result to the underlying file access path, and reduces the success of the completion from **:old** to **:new**. Since the code doesn't look at the VC file to see if the specified branch is already defined, it can't legitimately return **:old**.

VC File Substrate Design and Implementation

Version Control models each file as a tree of versions, with each version made up of an ordered list of sections. All of the versions are stored in a single text file. The file substrate provides an interface to manipulate these files. In theory, this interface is independent of the stored representation. In practice, it probably includes some dependencies on the current text file representation which will have to be ironed out in the process of designing a Statice representation.

The source files of the major components of the file substrate will be mentioned. In addition, however, the file `version-control:vc;defs.lisp` contains common definitions for the file substrate.

Common Structures and Functions for the VC File Substrate

There are some structures and types that are used throughout the file substrate and its applications. Note that some of these structures represent the "same" information as other structures returned by the file parser. The parser representation

is chosen to precisely reflect what's in the file. These representations exists to avoid dependencies between the current file format and the application interface to encached files. See the section "VC File Parser".

vci:file-version *Structure*

This structure is the basic internal specification of a file version. it has two slots: the branch name and the branch version. Branch names can be strings, and branch versions can be positive integers or one of the following:

:newest the highest numbered version in the branch.

:oldest the lowest numbered version in the branch.

:parent the parent version of the oldest version in the branch. This will be in *a different branch*. See the section "VC Pathnames Base Implementation".

vci:file-versions-equal *v1 v2* *Function*

Returns **t** if v1 and v2 name the same file version.

vci::file-version-lessp *v1 v2* *Function*

Provides a standard sort order for file version. The ordering is **string-lessp** for the branch name. Within the same branch, symbolic versions come first, and the numeric versions sort in numeric order.

vci:file-version-info *Structure*

This structure includes **version-control-internals::file-version** and adds some additional information. Since it includes **version-control-internals::file-version** it is a subtype of that, and so any interface that requires a **version-control-internals::file-version** will also take one of these. The slots of this structure are as follows:

parent-version for any version except the root version of the file, a **version-control-internals::file-version-info** for the parent version.

trailer a reference to a **version-control-internals::file-version-trailer** structure for this version. This can by **nil** if there was no trailer stored for the version in the file. Note that this is *not* the same as a **version-control-internals::parsed-file-trailer** returned by the file parser. See the method **(flavor:method :read-trailer-item vci::parser)**.

length the length, in bytes, of this file version as recorded by the program that created it. This is not guaranteed to be accurate, and is only maintained to allow for file stream progress notes. See the section "VC File Header".

author — The user who wrote the version, as recorded by the program that created it. This is not guaranteed to be accurate. See the section "VC File Header".

creation-date — The creation date of this version as a universal time. See the section "VC File Header".

vci::file-version-trailer *Structure*

This structure is the standard in-virtual-memory representation of a version trailer. Note that it is not the same as **version-control-internals::parsed-file-trailer**, which is the representation returned by the parser. See the method **(flavor:method :read-trailer-item vci::parser)**.

description — A string describing the entire file version. Generally it summarizes all of the changes to sections in this version.

per-section-array — An array of **version-control-internals::file-version-trailer-per-section** structures. These are created at the discretion of the program that creates the file version. In particular, there won't necessarily be a per-section structure for each changed structure. Typically, there is a per-section structure for each changed section for which the user provided explicit modification comments. See the structure **vci::file-version-trailer-per-section**.

vci::file-version-trailer-per-section *Structure*

This structure represents trailer information in virtual memory for a single changed section. Its conc-name is **file-version-trailer-ps-**.

The slots are as follows:

section-id — The section number of the section.

description — A string describing the changes to this section.

vci:section-boundary-blip *Structure*

This structure is used to represent a transition between two difference sections of a file. Interfaces that return the text of a file version return these on request from applications. The conc-name for this structure is `sbb-`.

The slots are as follows:

begin-section-id — The section number of the section which starts with the next text record returned.

Common Conditions for the VC File Substrate

The file substrate conditions are important for two reasons. First, they allow code that handles conditions for ordinary files (e.g. **fs:file-not-found**) to work on version control files. Second, the interfaces to the substrate do not have keyword arguments to return **nil** on errors. They just signal. Programs that want to tolerate error situations have to be prepared to handle conditions.

vci:non-version-controlled-file *Flavor*

This condition signalled whenever a function that operates on version controlled files is called with an ordinary flat file.

The following methods are relevant:

version-control-internals::non-version-controlled-file-pathname
Returns the pathname of the offending file.

vci::undefined-file-version *Flavor*

This condition is signalled when a reference is made to an undefined version of a version controlled file. It is based on **fs:file-not-found**.

The following methods are relevant:

sys:proceed method **:new-version**
permits the handler to supply an alternative version.

version-control-internals::undefined-file-version-encached-file
returns the encached file or encached file header in question.

version-control-internals::undefined-file-version-undefined-version
returns the offending version.

vci::undefined-file-branch *Flavor*

This condition is signalled when a reference is made to an undefined branch of a version controlled file. This is never used by interfaces that take file versions as arguments, only by those that take branch names.

The following methods are relevant:

sys:proceed method **:new-branch**
permits the handler to supply an alternative branch.

version-control-internals::undefined-file-version-encached-file
returns the encached file or encached file header in question.

version-control-internals::undefined-file-version-undefined-branch
returns the offending version.

vci:duplicate-file-branch *Flavor*

This condition is signalled on an attempt to create a second branch with the same name as an existing branch of a version controlled file.

The following methods are relevant:

version-control-internals::duplicate-file-branch-encached-file
returns the encached file.

version-control-internals::duplicate-file-branc-branch
returns the offending branch name.

VC File Format

A version control file is made of three parts: the header, the text, and the trailer. The organization meets several requirements aimed at optimizing compiling source. When compiling, the goal is to read the text of the newest version in some branch as efficiently as possible in both space and time. This translates into the following concrete requirements:

- A program can read out all of the text for an particular version while reading the file in a strictly forward direction, that is, it should not be *required* to encache any significant fraction of the file in virtual memory to read out one version. This is intended

- A program can read only the beginning of the file and get enough information to list its contents. This includes the author, creation date, and length of each version.

- Any information not needed to compile any version must be at the end, and must not need to be read at all when reading out a single version.

In addition, the file format is designed to be extensible. Specifically, the different parts of the file are separated by delimiters with a standard format, so that a program can extract one part without complete knowledge of the rest of the format. The delimiters are lines that begin with the character π (pi). Doubling is used when data could contain a π. There are three kinds of pi records:

π* - one liner a one liner is a control record that is self-contained.

πB TAG - begin group
a πB record begins a group which is ended by the matching πE.

πE TAG - end group
a πE record ends a group started with a πB.

The file format consists of three parts: the header, the text, and the trailer.

In the file, this looks like:

```
-*- Version-Control: 2; -*-
πB VTB number_of_versions
version-table-entry-1
...
version-table-entry-N
πE VTB
π* PROPERTIES
#S(properties of this file.)
πB TEXT number-of-sections
file-body
πE TEXT
πB FTR
trailer blocks
πE FTR
```

The header begins with the start of the file and ends with the beginning of the text. Then comes the text, and then the file trailer.

VC File Header

The file header contains the attribute list, the version table, and the property list. The Version-Control attribute identifies the file as a version controlled file. The value of the attribute is the file format version, to allow upward compatible support of new file versions. No provisions have been made for downward compatibility of file versions. The code assumes that it cannot handle any version that it dosen't know about.

The version table is an ordered list of versions, with one line per version. Each version is implicitly numbered by its position in the list, with the first version being version one. Version zero is reserved as a marker and is never used. These versions are called *internal versions* to distinguish them from external versions like `"Initial.12"`.

Each version-table-entry looks like:

```
parent-version-number branch-name branch-version length author date

  or

*
```

If the line contains only a `*`, that means that the version has been logically deleted from the file. The placeholder keeps all the other versions in the same position in the table, which simplifies the code that implements deletions.

The parent version number is zero for the root version of the file. If the branch name is `""`, it means that this version is in the same branch as its parent. The

length, author, and (creation) date are present so that the stream that reads one version out of a file can support the standard stream protocols which return those items. The length is the length in bytes of this version's text image. The file format dosen't care what this length is; it has no effect on its operation. It is only there to be returned when something sends a stream a **:length** message. It is not guaranteed to be accurate, and is only maintained to allow for file stream progress notes.

The property list follows the version table. The property list is an arbitrary lisp property list. It is used to allow version control tools to record arbitrary information about a file without having to make specific provisions in the file format. While this is modular and convienient, it is a performance problem because the lisp reader is extraordinarily slow.

For extensibility, the properties are not written as a simple list, but rather as the structure **version-control-internals::encached-file-properties-1**. This structure has a single slot which contains the property list itself. If the format of the property list is ever changed, a different structure can be written without ambiguity.

VC File Text

The text section contains the images of all of the sections. The sections are stored in the order that they are found in the file versions. What, you may ask, if the order has changed from one version to another? Then the section will occur as many times as needed so as to appear in the right place in all versions.

A section begins with:

```
πB FS section_number
```

and ends with

```
πE FS section_number
```

No well-formed file will ever have anything after a `πE FS` and before the next `πB FS`. Within the section delimiter records are text records and control records. The control records specify text to be inserted and/or deleted to construct a particular version of the section.

insertions and deletions begin with:

```
πB IN version_number   πB DL version_number
```

and end with

```
πE IN version_number   πE DL version_number
```

Any text that is not inside any insertions or deletions is text for version 1, by convention. To construct the text for a particular version **v**, a program proceeds as follows, starting from the beginning of the section and considering each line in turn:

- Any text found before the first insert or delete is included, since version 1 is the root of all versions in the file.

- At a πB IN, if the version_number is **v** or an ancestor of **v**, then continue processing. Otherwise, skip everything (including π records) up to the matching πE IN.

- At a πB DL, if the version_number is **v** or an ancestor of **v**, then skip everything up to the matching πE DL. Otherwise continue processing.

- Ignore any πE IN or πE DL records found while not skipping for a match.

If the scan for text for the desired version finds all text for that section has been deleted, then the section is ignored. For any particular version of the file, each section will be nonempty no more than once, representing its position in the order of sections for that version. It will be empty for all the places that it turns in elsewhere in the order in other versions.

VC files are full character files. In general, this is transparent, with fat character encoding handled at a lower level. One exception is diagram lines. The diagram storage protocol returns diagram instances from **:line-in**. Programs that interpret VC files have to enable diagram lines (with the **:set-return-diagrams-as-lines** message) and pass them back as text lines.

VC File Trailer

The file trailer contains additional information about the file versions that is not needed to reconstruct the text images of the versions. The trailer is organized as a series of blocks, at most one per version.

The overall structure of the trailer section is:

```
πB FTR
file-version-trailer-1
...
file-version-trailer-N
πE FTR
```

Each trailer consists of:

```
entry-length-in-chars
internal-version-number
"general description" n-sections
section-id-1 "per-section-comment-1"
...
section-id-n-sections "per-section-comments-n-sections"
π*
```

Unlike the header and the text, the trailer is *not* purely organized by lines. Each entry starts with a count of the characters in that entry. The π* lines allows for consistency checking; if there isn't a π* after the specified number of characters, then the file is broken. The general description is user commentary on the entire file version. Then there are zero or more per-section entries that contain comments on individual sections.

VC File Parser

VC files are parsed by instances of the flavor **version-control-internals::parser**. This flavor encapsulates a stream open to a file, and returns data structures built from the contents of the file. The parser is not responsible for selecting the text of a particular version from the file, because some applications need one version, while others want all of them. Rather, the parser is concerned with translating from text to lisp.

The parser is defined in `version-control:vc;file-parser.lisp`.

vci::parser *Flavor*

The parser flavor is initialized with two keyword arguments, **:stream** and **:file-version**. The stream must be a stream open to a VC file positioned *after* the attribute list. This is because typical applications of the parser open the file, read the attribute list, and only call version control when the Version-Control attribute is present.

The file-version must be the value of the Version-Control attribute. This permits the parser to support multiple file versions. At this time, it only supports version 2.

Once the parser is instantiated, it supports a series of messages *in order*. First **:read-header** reads the header. Then **:read-text-item** reads each record of the text section. Finally, **:read-trailer-item** reads each trailer.

(flavor:method :read-header vci::parser) *Method*

This message returns a structure (**version-control-internals::parsed-file-header**) containing the information read from the file header. This message must be the first sent to the parser instance.

The structure returned is defined as follows:

```
(defstruct (parsed-file-header
             (:conc-name pf-header-)
             (:constructor make-pf-header)
             (:print-function print-pf-header)
             )
  (version-info-array nil)
  (n-versions 0)
```

```
  (n-sections 0)
  (properties nil))
```

The **version-control-internals::pf-header-version-info-array** slot contains an array of **version-control-internals::parsed-file-version-info** structures, one for each version.

This structure is defined as follows:

```
(defstruct (parsed-file-version-info
             (:conc-name pf-version-info-)
             (:constructor make-pf-version-info)
             )
  (deleted-p nil)
  (parent-version 0)
  (branch-name nil)
  (branch-version 0)
  (length 0)
  (author "")
  (creation-date 0)
  )
```

The slots in these structures correspond directly to the file format. Note that the **version-control-internals::pf-version-info-deleted-p** slot indicates a version that was marked with a `*`.

(flavor:method :read-text-item vci::parser) *Method*

This method may only be used after **:read-header**. It returns a single text section record or **:end-of-text**. Text section records come in three Common Lisp types:

string an ordinary text record.

instance a diagram instance, treated as a text record.

version-control-internals::encached-control-record
a control record, either πB FS, πE FS, πB IN, πE IN, πB DL or πE DL. These are represented as encoded fixnums.

VC Encached Control Records

An encached control record is a fixnum that represents a control record in the text section of a VC file. Fixnums are used to avoid consing a structure for each record, of which there are many in each file.

The top eight bits of the control record **version-control-internals::cr-type** represent the type of the record. Named constants are provided in `version-control:vc;defs.lisp` for the types. The remaining 16 bits have different definitions depending on the type. For the

beginning and end of a section, those bits contain the section number. For the beginning and end of inserts and deletes, those bits are the relevant version number. Thus, **version-control-internals::cr-version** and **version-control-internals::cr-section** access the same bits of the fixnum.

(flavor:method :read-trailer-item vci::parser) *Method*

This method returns **version-control-internals::parsed-file-trailer** structures for the trailer blocks in the file, and then **:end-of-trailers** after the last one.

version-control-internals::parsed-file-trailer is defined as follows:

```
(defstruct (parsed-file-trailer
             (:conc-name pf-trailer-)
             (:constructor make-pf-trailer))
  (version-number)
  (description)
  (per-section-list))

(defstruct (parsed-file-trailer-per-section
             (:conc-name pf-trailer-ps-)
             (:constructor make-pf-trailer-ps))
  (section-id)
  (description))
```

These slots are parallel to the items in the textual file format.

VC Encached Files and Encached File Headers

Encached files are the interface to VC files for applications that do more than just read the text of a single version of the file. An encached file is an in-virtual-memory representation of the contents of a file. There are two kinds of encached files: full encached files (called "encached files") and encached file headers. An encached file header contains only the information from the header of the VC file.

A central registry of interned encached files and encached-file-headers is maintained for programs that want to share a cache. The editor always uses the registry, and other programs that want to permanently encache files use it as well. Programs that read many VC files that the user is unlikely to edit any time soon don't use the cache to avoid filling virtual memory.

VC Encached File Headers

Encached file headers are used by programs that need to find out about the versions defined in a file but don't expect to process more than one, or even any, of the textual data. The flavor **version-control-internals::encached-file-header** provides the representation for encached file headers. See the flavor **vci::encached-**

file-header. **version-control-internals::encached-file-header** has no init options. Note that for many applications it is appropriate to use the function **version-control-internals::find-or-make-encached-file-header** rather than to just **(make-instance 'vci:encached-file-header)**.

Functions for Use with Encached File Headers

vci:branch-defined-p *encached-file-header branch-name* *Function*

Given the name of a branch, returns T if there are any versions in the file whose branch name the supplied name. Otherwise it returns NIL.

vci:branch-last-version *encached-file-header branch-name* *Function*

Given a branch name, returns the **version-control-internals::file-version-info** structure for the newest version in the branch.

vci::encached-file-branch-parent-version *encached-file-header branch-name* *Function*

Given a branch name, returns the **version-control-internals::file-version-info** structure for the parent version of the first version in the branch. That is, if a user started with version `Main.12`, read it into the editor, modified it, and saved it out as `Whatever.0` (a new branch), then the parent version of branch `Main` is 12.

vci:encached-file-header-merge-version *encached-file-header version* *Function*

Given a **version-control-internals::file-version** structure, returns a **version-control-internals::file-version-info** structure with **:newest**, **:oldest**, or **:parent** replaced by the specific version referenced. In the case of **:parent**, the resulting version will have a different branch name.

vci::encached-file-name *encached-file* *Function*

Returns the name of the encached file, if any. Usually this is its pathname.

vci:encached-file-pathname *encached-file* *Function*

Returns the pathname associated with the encached file, if any.

vci::encached-file-per-version-info *encached-file-header internal-version* *Function*

Returns the internal **version-control-internals::per-version** structure for the specified internal version. This is a side door that is used by parts of the file substrate that aren't methods of **version-control-internals::encached-file-header**. In fact,

the only caller is the flavor **version-control-internals::encached-file-version-info**, whose independent existence is not necessarily a good thing.

vci::encached-file-read-header-from-parsed-header *encached-file-header header* *Function*

Given a **version-control-internals::parsed-file-header** structure, initializes an **version-control-internals::encached-file-header** structure to reflect that header. This is used to reuse the same **version-control-internals::encached-file-header** instance to process more than one file.

vci::encached-file-stored-property-list *encached-file-header* *Function*

This returns the stored property list for the file. It can be used with setf (and especially in cliches like **(setf (getf (encached-file-stored-property-list ef) :foo) :bar)**) to side-effect the property list. Note that changing the property list has no effect until and unless the encached file is written back out to the text file.

vci:encached-file-version-info *encached-file-header version* *Function*

Given a **version-control-internals::file-version** structure, returns a **version-control-internals::file-version-info** structure for the version.

vci:encached-file-versions *encached-file-header* *Function*

This returns an array containing one **version-control-internals::file-version-info** structure for each version defined in the file.

vci:leaf-file-version-p *encached-file-header version* *Function*

Returns T if the supplied **version-control-internals::file-version** is a leaf version in the file. That is, if it is the newest version in its branch. Note that this function returns information based on the contents of the encached-file-header structure. It does *not* check to see if there are newer versions out on disk or in an editor buffer.

vci::version-defined-p *encached-file-header version* *Function*

Returns T if the specified **version-control-internals::file-version** is defined in the file, and NIL otherwise.

vci:encached-file-ancestor-version-p *encached-file-header version putative-ancestor-version* *Function*

Given a file-version and a file-version which might be its ancestor, returns **t** if the putative ancestor is in fact an ancestor and **nil** otherwise.

vci:encached-file-parent-version-p *encached-file-header version putative-parent-version* *Function*

Given a file-version and a file-version which might be its parent, returns **t** if the putative parent is in fact the parent and **nil** otherwise.

Implementation of VC Encached File Headers

Data Structures of VC Encached File Headers

There are two data structures relevant to encached file headers: the flavor **version-control-internals::encached-file-header** itself, and the **version-control-internals::per-version** structure.

vci::encached-file-header *Flavor*

This flavor encaches the header of a VC file. Unlike the defstruct **version-control-internals::parsed-file-header**, this conses auxiliary information that accelerates common operations, and provides methods for operations on the header.

The instance variables are as follows:

version-table — The table describing all of the versions defined in the file. See the section "VC Encached File Header Version Table".

stored-property-list — The file's stored property list, accessed with the function **version-control-internals::encached-file-stored-property-list**. See the function **vci::encached-file-stored-property-list**.

VC Encached File Header Version Table

An array of **version-control-internals::per-version** defstructs, one per version. Index zero of this array always contains NIL, so that internal version *n* is always found in **(aref version-table *n*)**. See the structure **vci::per-version**.

vci::per-version *Structure*

This structure represents the data about each version of the file. Some of the fields are unused within the flavor **version-control-internals::encached-file-header** and only used by **version-control-internals::encached-file**. All of the fields are described here. The conc-name of this structure is pv-. They are constructed with the function **version-control-internals::make-per-version**.

The slots are as follows:

internal-version — The internal version number. **(pv-internal-version (aref version-table *n*))** will always be **n**.

parent-version — The internal version number parent version of this version. The root version (currently always version 1) has zero in this slot.

branch-name — The string branch name of the external version.

branch-version — The integer branch number of the external version.

ancestor-bitmap — A boolean array. For version *n*, this array is of length *n*. For each lower-numbered version, this array contains **t** if the version is an ancestor of this version, and **nil** otherwise. This is used when processing text information for the version to resolve insert and delete groups. This field is never used in **version-control-internals::encached-file-header**, only in **version-control-internals::encached-file**. This field is only filled the first time that it is needed.

leaf-p — **t** if this version is newest in its branch, **nil** otherwise.

trailer — A **version-control-internals::file-version-trailer** structure for this version. See the structure **vci::file-version-trailer**. This is here to avoid consing a new one every time an application asks.

version-info — A **version-control-internals::file-version-info** structure for this version. See the structure **vci:file-version-info**. This is here to avoid consing a new one every time an application asks.

version-reconstruct-path — An array of **version-control-internals::reconstruct-entry** structures that represents the order of sections in this versions.

Internal Functions and Methods of VC Encached File Headers

An **version-control-internals::encached-file-header** is primarily a data structure. The only code internal to it is the method that converts an external version to an internal version number by searching the version table, **version-control-internals::lookup-external-version**.

vci::lookup-external-version *encached-file-header version* &key *error-p* — *Function*

Returns the internal version integer for a **version-control-internals::file-version** structure. This is an internal interface of the encached-file-header, and should probably be a defun-in-flavor.

(defun-in-flavor vci::consing-with-instance vci::encached-file-header) &body *body* — *Flavor Internal Macro*

This macro is used throughout **version-control-internals::encached-file-header** and **version-control-internals::encached-file** to maintain locality between the vari-

ous structures that are allocated. This macro binds **version-control-internals::default-cons-area** to the area of **self**.

VC Encached Files

Encached files provide the in-virtual-memory representation of an entire VC file. They provide interfaces for extracting the text of version, creating new version, and deleting versions.

Encached files are implemented by the flavor **version-control-internals::encached-file**. Applications can create instances of **version-control-internals::encached-file** with **make-instance**. However, for in many circumstances it is preferable to call **version-control-internals::find-or-make-locked-encached-file** and interact with the global interned cache of encached file.

Making a VC Encached File

To get an encached file for a VC file via the global cache, call **version-control-internals::find-or-make-locked-encached-file**. To make an encached file instance independent of the cache, make an instance with **make-instance**. The init options for encached file are optional.

Once you have an instance, you use the function **version-control-internals::read-in-file** to read a VC file into the instance. You can reuse the same instance over and over again.

(flavor:method :pathname vci:encached-file) *pathname* *Init Option*

Supply a pathname for the encached file. For **version-control-internals::encached-file**, this pathname is only used in the **sys:print-self** method to make it easier to sort out different encached file.

(flavor:method :name vci:encached-file) *name-string* *Init Option*

This provides a name string for printing the encached file object. The default is to use the pathname as the name.

Functions for Use with VC Encached Files

vci:add-new-version *encached-file parent-version external-version* δ*-section-array trailer* *Function*

This method is the external interface to creating a new version in an existing file. It cannot create new root version, but only versions that modify existing versions. Thus it takes a parent-version. The external-version is a **version-control-internals::file-version** for the version to be created. δ-section-array is an array of

version-control-internals::δ-section structures. There must be at least one for each section in parent-version. See the structure **vci:δ-section**. trailer is a **version-control-internals::file-version-trailer** structure, and is stored as the trailer for the new version.

vci::clone-file-version *encached-file old-external-version new-version-info trailer* *Function*

Creates a new version in which nothing has changed. This is used to establish a new branch of the file independently of actually changing any text in the new branch.

vci::delete-encached-file-branch *encached-file branch-name* *Function*

This removes all versions of a specified branch from the encached file. If any versions have parents that are deleted those versions are changed to use the parent of the deleted branch as their parent, and their inserts and deleted are recalculated accordingly.

vci::delete-versions *encached-file version-list* *Function*

This function deletes all of a specified list of versions from the encached file. If any version not deleted have one or more of the deleted versions as ancestors, then they are changed to use the last un-deleted ancestor of the deleted version as an ancestor. You can't delete the root version of the file.

Note that this function does not provide concurrency control on **version-control-internals::locked-encached-files**. It should be called from within a **version-control-internals::with-locked-encached-file-locked**.

vci::encached-file-per-section-info *encached-file section-x* *Function*

Returns the **version-control-internals::section-info** structure for the specified section number. This is a side-door interface used only for **version-control-internals::encached-file-version-info** instances.

vci:encached-file-section-not-empty-for-version *encached-file section-number version* *Function*

Returns T if the specified section has any text in it for the specified version. An empty section is considered deleted by convention, however, in some circumstances a section can be deleted from one version and then reappear in one of its successors.

vci::encached-file-truename *encached-file* *Function*

If the encached file has a pathname, the truename is encached and available via this function.

:initialize-from-δ-section *δ-section-array version trailer* *Message*

This message is used to initialize an empty encached file from scratch, as opposed to from a VC file in the file system. version names the root version. δ-section-array is an array of **version-control-internals::δ-section** structures that specifies the initial contents. See the structure **vci:δ-section**. trailer is the trailer for the root version. This should be converted to a generic function.

vci::make-retrieve-continuation *encached-file version* &key *section-marks one-section start-section end-section* *Function*

This function is the fundamental interface for retrieving text from an encached file. It returns a function which when called with no arguments returns each of the lines of the text in turn. After the last line it returns **:eof**. There is a **loop** path that provides a more convienient interface atop this one.

This design allocates a new closure on the heap every time that a program retrieves text from an encached file. The alternative would be an interface that takes a downward function and calls it with each record in turn. One problem with that is that the current **loop** implementation can't handle it.

The arguments to the function are as follows:

version	The version of the text to retrieve.
section-marks	When retrieving text for more than one section, this controls whether section boundary blips are returned to mark the boundaries. If this is **t**, then a **version-control-internals::section-boundary-blip** will be returned instead of a record at each section boundary.
one-section	If this is non-**nil**, it must be an integer section number. The function returned will return each of the lines of that section.
start-section	specifies the first section to return. If this is **nil**, the retrieval starts at the beginning of the file.
end-section	specifies the last section to return. If this is **nil**, the retrieval continues through the last section of the file.

Loop Paths for Retrieving Information From VC Encached Files

There are two loop paths defined for retrieving text information from encached files: **text-records** and **text-and-section-records**. **text-records** returns the lines (and diagram lines) for one or more sections of a specified version. **text-and-section-records** returns the lines (and diagram lines) for one or more sections of a specified version, and inserts **version-control-internals::section-boundary-blip**s to

indicate the transitions between sections. See the structure **vci:section-boundary-blip**.

The loop keywords that control these paths are as follows:

of	Specifies the encached file for the retrieval. This is required.
in-version	Specifies the **version-control-internals::file-version** for the retrieval. This is required.
only-section	Specified the section number for retrieval of the text for a single section.
from-section	Specifies the section number of the first section to be retrieved. The default is the first section defined in the file version.
to-section	Specifies the section number of the last section to be retrieved. The default is the last section defined in the file version.

:number-of-sections *Message*

This message, sent to an encached file, returns the total number of sections defined in all versions. It is the highest section number defined for any version.

vci::read-in-file *encached-file stream* *Function*

This function initializes an encached file by reading a VC file from the file system and representing it in virtual memory. The stream should be an element type **character** stream positioned at the beginning of the file.

vci::read-in-encached-file *pathname* &key *area encached-file* *Function*

This function reads a VC file from a file system file into an encached file. It provides standard proceed options for retry in case of network problems. If the encached-file argument is **nil**, then a new encached file instance will be consed in the area specified by the area argument, with the default of **version-control-internals::default-cons-area**. Note that for many applications **version-control-internals::find-or-make-locked-encached-file** is more appropriate.

:reconstruct-file *stream* *Message*

This message to an encached file writes out the current contents as a VC file to the stream provided.

vci:encached-file-version-section-order *encached-file version* *Function*

This function returns a list of the section numbers of the sections defined in the specified file version in order.

Implementation of VC Encached Files

VC Encached File Data Structures

An encached file is an encached file header with additional information to represent the text and the trailers. The trailers are easy: they are represented by **version-control-internals::file-version-trailer** structures referenced by the **version-control-internals::per-version** structures in the version table.

The text is more complex. In the VC file, each section can occur more than once so that any version can be read while reading the file monotonically. See the section "VC File Text". In the encached file, all the occurrences of each section are brought together into a single **version-control-internals::section-info** structure. (Unfortunately, the word "occurrence" is mispelled "occurence" consistently in the code.) The **version-control-internals::section-info** structures are organized in the section-info instance variable of **version-control-internals::encached-file**. The order of the occurrences in the VC file, and thus the order in which they must be scanned to reconstruct the text of any particular version, is recorded in the reconstuct table.

vci:δ-section *Structure*

This structure is used to describe a section when creating a new version of an encached file. The interfaces that make new versions take as a argument an array of these structures. It specifies the disposition of each section in the parent file version, plus any new sections, in order.

It has the following slots:

section-id
: The section number of the section under consideration. For new sections, this is :new on input to the interfaces that create new versions, and is changed to the assigned section number on output.

new-version-interval
: If the section is unchanged from the previous version, this contains **nil**. If it has been changed, this contains a ZWEI interval with the new text. If the section is deleted altogether, this contains **:deleted**.

vci:encached-file *Flavor*

This flavor encaches the entire contents of a VC file. It is based on **version-control-internals::encached-file-header**, and so inherits all of its methods.

The instance variables are as follows:

section-table
: an array of **version-control-internals::per-section** structures, one for each section defined in the file, indexed by section number.

reconstruct-table	an array of **version-control-internals::reconstruct-entry** structures, one for each occurrence of a section in the file.
queue	a process queue, used to interlock modifications to the encached file between processes.
name	a string for **sys:print-self**, or **nil**.
pathname	a pathname or **nil**. Used as the default name.
truename	if the pathname is not **nil**, the truename.

In addition, the version-table instance variable is inherited from **version-control-internals::encached-file-header**.

VC Encached File Header Version Table

An array of **version-control-internals::per-version** defstructs, one per version. Index zero of this array always contains NIL, so that internal version *n* is always found in **(aref version-table *n*)**. See the structure **vci::per-version**.

vci::per-version *Structure*

This structure represents the data about each version of the file. Some of the fields are unused within the flavor **version-control-internals::encached-file-header** and only used by **version-control-internals::encached-file**. All of the fields are described here. The conc-name of this structure is pv-. They are constructed with the function **version-control-internals::make-per-version**.

The slots are as follows:

internal-version	The internal version number. **(pv-internal-version (aref version-table *n*))** will always be **n**.
parent-version	The internal version number parent version of this version. The root version (currently always version 1) has zero in this slot.
branch-name	The string branch name of the external version.
branch-version	The integer branch number of the external version.
ancestor-bitmap	A boolean array. For version *n*, this array is of length *n*. For each lower-numbered version, this array contains **t** if the version is an ancestor of this version, and **nil** otherwise. This is used when processing text information for the version to resolve insert and delete groups. This field is never used in **version-control-internals::encached-file-header**, only in **version-control-internals::encached-file**. This field is only filled the first time that it is needed.
leaf-p	**t** if this version is newest in its branch, **nil** otherwise.

trailer — A **version-control-internals::file-version-trailer** structure for this version. See the structure **vci::file-version-trailer**. This is here to avoid consing a new one every time an application asks.

version-info — A **version-control-internals::file-version-info** structure for this version. See the structure **vci:file-version-info**. This is here to avoid consing a new one every time an application asks.

version-reconstruct-path — An array of **version-control-internals::reconstruct-entry** structures that represents the order of sections in this versions.

vci::section-info *Structure*

This structure is an array leader. The body of the array is a sequence of text and control records.

The slots are as follows:

n-records — the number of text and control records in the section. This is in fact the fill-pointer.

occurence-count — the number of occurrences in the section. See the section "VC Encached File Reconstruct Table".

In the VC file, a given section can occur more than once to represent the different places that it appears in the order of sections in different versions. In the encached file, each section has a single **version-control-internals::section-info** structure containing all of the occurrences. See the section "VC Encached File Reconstruct Table".

VC Encached File Reconstruct Table

In the VC file, a given section can occur more than once to represent the different places that it appears in the order of sections in different versions. In the encached file, each section has a single **version-control-internals::section-info** structure containing all of the occurrences. The array will look like the following:

```
πB FS 1
πB DL 2
this is a line
this is another line
πE DL 2
πE FS 1
πB FS 1
πB IN 2
this is a line
this is another line
πB IN 2
πE FS 1
```

Each BFS/EFS pair surrounds an occurence.

The reconstruct table is a sequence of **version-control-internals::reconstruct-entries** that specifies the order that the sections' occurrences appear in the VC file. Since each section is only nonempty once per file version, walking the occurences as specified by the reconstruct table will produce the text for the entire file version in the correct order.

vci::reconstruct-entry *Structure*

This structure specified an entry in the reconstruct table. If has the following slots:

section-number	which section comes at this point in the reconstruct order.
occurence	which occurrence of the section. The first occurrence is zero. (The slot is in fact mispelled in the code.)

Initialization of VC Encached Files

Encached files are initialized in one of two ways: by reading a VC file, or by creating a new root version from a set of ZWEI intervals. The function **version-control-internals::read-in-file** reads a VC file from a stream and initializes an encached file to represent it. The message **:initialize-from-δ-section** initializes an encached file from an array of **version-control-internals::δ-section** structures.

Retrievals from VC Encached Files

version-control-internals::make-retrieval-continuation implements retrieval. It consists of two complex lexical closures. The simpler one returns the text lines of a single section for a particular version. The complicated one works for multiple versions.

The code structure is the co-routine that you would expect, with lexical state being used to remember the location in the file/section of the last record returned.

This choice of implementation is questionable, since it forces consing of a lexical closure for each retrieval. If they don't get EGC'ed, then they could build up in a hurry.

There is no interlocking to prevent some other process from modifying the file in the middle. There are two bad reasons for this. First of all, the use of an upward closure fails to provide a scope for locking. Second of all, there is no intra-machine locking substrate with multi-reader locks.

Making New Versions of VC Encached Files

The most complex function of an encached file is to create a new version. The interface for creating a new version is **version-control-internals::add-new-version**.

It takes an encached file, a parent version, a specification of the new text, and the trailer.

The specification of the new text is an array of **version-control-internals::δ-section** structures. There must be one such structure for each section in the parent version, plus additional structures for any new sections. The order of structures specifies the order of sections.

This interface makes no constraint on the origin of the text for the new version. The text usually will come from some manipulation of the text of the parent version, but it dosen't have to. **version-control-internals::add-new-version** will match up sections and compute text differences irregardless.

(flavor:method vci:add-new-version vci:encached-file) *parent-version external-version δ-section-array trailer* *Method*

This function is the top-level driver for adding a new version to an existing file. It is responsible for reconciling the order of sections in the parent version with that specified for the new version.

Where a section is in the same place in the old version as the new version or is deleted in the new version, it simply uses **version-control-internals::δ-one-section** to do the work. When a section has moved, it has to delete the old occurence and create a new one. **version-control-internals::move-section** makes the changes to the section-info. When the relative order of two sections has changed, it moves the one that move the smaller distance in the order, since that changes less.

(flavor:method vci::δ-one-section vci:encached-file) *parent-version new-version parent-re δ-section-info* *Method*

This function is a subroutine of **version-control-internals::add-new-version** that processes a single section. parent-re is the **version-control-internals::reconstruct-entry** for this section in the parent version, and specifies the section and the occurence in the section that is being changed. δ-section-info is the **version-control-internals::δ-section** for this section by the caller of **version-control-internals::add-new-version**.

If δ-section-info specifies that the section is to be deleted, (**(eq (δ-section-new-version-interval δ-section-info) :deleted)**), then this calls **version-control-internals::delete-section** to do the work. Otherwise, it walks down the text of the old version in the encached file and the new version in the supplied interval, creating insert and delete groups as needed. Its comparison algorithm is simple: it considers itself resynchronized when it finds one matching line for the two versions. There is no provision for deciding that the new version of a section is wildly different from the old version, and just wrapping a delete around the entire old copy and an insert around the new. This can lead to a proliferation of control records, and is an area that needs work.

VC Locked Encached Files

A locked encached file is an encached file with network (inter-host) concurrency control. The concurrency control serves two purposes:

File Update — When adding to or deleting from a file, a locked encached file locks out other hosts while it ensures that the most recent copy is read in, adds the new version, and writes it out.

Branch Update — As a service to the editor, a locked encached file will hold a lock on a branch of the file. This allows one user to claim the right to add the next version to a particular branch and prevent any other user from doing the same thing.

All locked encached files have pathnames, since the pathname is used as a locking handle amongst hosts.

Users of locked encached files must make provisions for the unreliability of the lock-simple network lock server. If the file server containing the VC file crashes, all of the locks are unlocked. This isn't a problem for file updates, because if the server is down you can't write out the new copy. For branch locks, though, a user can lock the branch and then silently loose the lock due to a file server crash. The editor version control code detects this and signals takes corrective action.

Locked encached files are managed by the global interned file cache. Otherwise, you could have two applications on the same host with different locked encached file locking against each other, which would be a waste of time and virtual memory. To get a locked encached file for a VC file use the function **version-control-internals::find-or-make-locked-encached-file**.

Functions for Use with VC Locked Encached Files

vci:with-locked-encached-file-locked (*locked-encached-file*) &body *body* *Macro*

- This macro works as follows:
- Gets a write (exclusive) lock on the entire file.
- Checks to see if the encached file contains the latest copy of the VC file. If not, it reads it.
- Runs the body.
- Releases the lock.

vci:make-new-version-from-δ-section-array *locked-encached-file parent-version new-version δ-section-array trailer* *Function*

This functions replaces the **:initialize-from-δ-section** message and **version-control-internals::add-new-version** function on locked encached files. For new files, this checks to make sure that no one else has created a file at the same pathname before proceeding. It should be called within a **version-control-internals::with-locked-encached-file-locked**.

vci:write-out-new-file-version *locked-encached-file author* *Function*

This function replaces the **:reconstruct-file** message for locked encached files. It writes out the encached file as a VC file to its pathname. The author argument is provided so that the editor can use its usual **zwei:non-daemon-user-id** protocol to avoid LISP-MACHINE as a author. Other applications can just use **(send si:*user* :pretty-name)**. This should be called inside the same **version-control-internals::with-locked-encached-file-locked** as the call to **version-control-internals::make-new-version-from-δ-section-array**, **version-control-internals::delete-versions**, **version-control-internals::delete-branch**, or any other operations that modify the encached file, including its stored property list.

vci:reread-encached-file *locked-encached-file* *Function*

This function rereads the encached file from the VC file. It is used to recover the file to a clean state when a change is interrupted. Applications that modify encached files should be written to be able to restart the modification from the beginning, so that they can have an error recovery loop that rereads the file and tries all over again.

vci:lock-branch-for-modification *locked-encached-file version* &key *new-p* *Function*

This functions gets a write (exclusive) lock on the a branch of the file. version must be a **version-control-internals::file-version**. The lock is obtained for that version's branch. new-p should be **t** if the the lock is to prevent other users from creating this branch, and **nil** if the branch exists and the lock is to prevent other users from adding a new version to the end. No "with-branch-locked" macro is provided, because branch locks are generally held for an extended period of time.

vci:ensure-encached-file-up-to-date *encached-file* *Function*

This function checks to see if a newer version of the VC file exists than the one what was read into this encached file (or encached file header). If so, it reads it. Note that unless this is called inside of a **version-control-internals::with-locked-encached-file-locked**, there is no protection from races between applications. Note that **version-control-internals::with-locked-encached-file-locked** always calls this before running the body.

vci:unlock-branch *locked-encached-file version* *Function*

This function releases a lock on a branch obtained with **version-control-internals::lock-branch-for-modification**.

vci:branch-lock-status *locked-encached-file version* *Function*

This function returns the current network lock status of the specified branch. See the documentation for **lock-simple::lock-status** for the format of the information returned.

VC Header Pseudo Encached Files

A **version-control-internals::header-pseudo-encached-file** is a particular flavor of encached-file-header used in the global cache. The name was poorly chosen. The differences between a header pseudo encached file and an encached-file-header are:

- it always has a pathname, which serves as the cache key.
- it has a **version-control-internals::parser** mixed in, so that it dosen't have to cons one when it is reread.
- it supports **version-control-internals::ensure-encached-file-up-to-date**, with the semantics of rereading the header if a newer VC file has been written.

The Global Cache of VC Encached Files and Encached File Headers

Reading a VC file into an encached file or even an encached file header takes a significant amount of time, and consumes a significant amount of virtual memory. If there is any significant chance that a file will get used more than one, its worth "interning" it when read so that it can be used again. This is the purpose of the global caches.

There are two caches: one for headers, and one for full files. Since an encached file is a perfectly valid encached file header, the functions that search for headers search the cache of files as well. If an application searches for a full file and there is a cached header, the cached header is removed in favor of the file.

The file cache stores **version-control-internals::locked-encached-file**s. The header cache stores **version-control-internals::header-pseudo-encached-file**s.

Functions for Use with the VC File Cache

vci:find-or-make-encached-file-header &key *stream pathname* (*make-ok* **t**) (*area* **version-control-internals::*encached-file-area***) *Function*

Given a stream *or* a pathname, looks for an encached file header or file in the global cache. If the file is not found in the cache, the result depends on make-ok. If it is **t**, a new header will be added to the cache and the file read into it. If it is **nil**, the function returns **nil**.

If the file is not a version control file, then the condition **version-control-internals::non-version-controlled-file** is signalled. See the flavor **vci:non-version-controlled-file**.

vci:find-or-make-locked-encached-file &key *stream pathname new-file-p* (*make-ok* **t**) (*area* **version-control-internals::*encached-file-area***) *Function*

Given a stream *or* a pathname, looks for an encached file in the global cache. If the file is not found in the cache, the result depends on make-ok. If it is **t**, a new encached file will be added to the cache and the file read into it. If it is **nil**, the function returns **nil**.

If the file is not a version control file, then the condition **version-control-internals::non-version-controlled-file** is signalled. See the flavor **vci:non-version-controlled-file**.

vci::unencache-file *pathname* *Function*

Removes a file, identified by pathname, from the cache of encached files. This should be used with care, since some program (like the editor) can retain a reference to an encached file after it is removed. This function is intended for debugging and recovery.

vci::unencache-header *pathname* *Function*

Removes a file, identified by pathname, from the cache of encached file headers. This should be used with care, since some program (like the editor) can retain a reference to an encached file header after it is removed. This function is intended for debugging and recovery.

vci::unencache-all-files *Function*

Empties the caches of encached files and encached file headers. This should be used with care, since some program (like the editor) can retain a reference to an encached file after it is removed. This function is intended for debugging and recovery.

vci::*all-locked-encached-files* *Variable*

This variable contains the cache of locked encached files. It is an alist from pathnames to **version-control-internals::locked-encached-file**s. The pathnames are canonicalized as follows:

```
(let ((canonical-pathnane
  (send (send pathname :new-pathname :version :newest :vc-branch nil
                       :vc-version nil))
        :translated-pathname))

  ...
  )
```

vci::*all-encached-file-headers* *Variable*

This variable contains the cache of encached file headers. It is an alist from pathnames to **version-control-internals::header-pseudo-encached-filess**. The pathnames are canonicalized as follows:

```
(let ((canonical-pathnane
  (send (send pathname :new-pathname :version :newest :vc-branch nil
                       :vc-version nil))
        :translated-pathname))

  ...
  )
```

VC Encached File Branch Registry

The encached file format stores very little information about a branch: its name, and any information in the trailer for its first version. Applications need more information to provide better interfaces for the user. This information is provided by the branch registry.

The branch registry is just a list of **version-control-internals::file-branch** structures stored in the stored property list of the encached file. Each structure describes a branch.

No code in the file substrate demands that a **version-control-internals::file-branch** be recorded for each branch defined in the file. It is up to applications to maintain consistency. The advantage of this is that applications can hide information in "unregistered" branches. The disadvantage is that inconsistencies are possible. It would be better all around if file branch recording was required for all new branches, but with a more extensible data structure that could handle more kinds of branches.

vci:file-branch *Structure*

This structure stores a description of a branch. It has the following slots:

name the name of the branch.

private-user-name — if the branch contains the private work of a particular users, that user's name is stored here. The intention is that user interfaces will by default hide private branches from other users to avoid clutter.

parent-file-version — the file version of the parent version of the first version in the branch.

new-versions-permitted-p — **t** is this branch is in active use. **nil** if it has been decomissioned, and user interfaces should prevent or discourage users from adding to it. Nothing uses this yet.

successor-version — if new-versions-permitted-p is **nil**, then this has is the file version that is the logical successor of the last version in the branch. For example, if a branch has been merged into another branch, this would indicate the version that resulted from the merge.

author — the author of the branch.

creation-date — the creating date of the branch.

vci::update-file-branch *encached-file file-branch* *Function*

This function updates the data for a branch. All of the data in file-branch is copied into the permanent copy on the file property list. This function assumes that the branch has been previously registered, and signals an error otherwise.

vci:record-file-branch *encached-file file-branch* &key (*update-ok* **nil**) (*new-ok* **t**) *Function*

adds a record of a branch to the registry. The data in the file-branch structure is copied, so that later modifications to the structure have no effect on what is recorded in the encached file.

If update-ok is nil, and the branch is already registered, an error is signalled.

If new-ok is nil, and the branch is not already registered, then an error is signalled.

vci:lookup-file-branch *encached-file file-branch-name* *Function*

This returns a file branch structure for the branch named by file-branch-name, or **nil** if none is recorded. This returns a copy of the record in the registry, so that to change the information recorded you must call **version-control-internals::update-file-branch**.

vci::un-record-file-branch *encached-file file-branch-name* *Function*

Removes the record for a file branch from the registry.

vci:file-branch-name-alist *encached-file* *Function*

Returns an alist from names to file usuable in user interfaces that like alists. The alist elements are conses, not lists, so the **version-control-internals::file-branch** structure in each one is in the **cdr**, not **second**.

Streams for Reading VC File Text

Version control provides stream interfaces for reading the text of VC files. There are two streams. Both read the text of a single file version from a VC file. One reads the text directly from a VC file in the file system, and the other reads the text from an encached file.

For an operation like compiling the file, it is not in general worth encaching the file in the global cache. Many file compilations, such as those resulting from SCT system compilation, compile files which are unlikely to be edited any time soon, and reference many files. It is more important to read a single version fast than to make all the version available in virtual memory. However, if some other application has already encached the file, there is no reason not to just use the encached copy.

Functions for Use with VC File Streams

The interfaces described here will open VC file streams. Calling **open** with a version controlled pathname returns VC file streams.

vci::open-one-version-file-stream *pathname* &optional *version* &rest *open-arguments* &key *only-section* &allow-other-keys *Function*

Returns a stream that reads the text of a file version of a VC file. Pathname should be the pathname of the VC file, not a version controlled pathname. If your application knows the file version in advance, it should supply a **version-control-internals::file-version** in version. If not, it can use **version-control-internals::vc-file-stream-encached-file-header** to get the encached file header, and use it to choose the appropriate version, and then do: **(setf (vc-file-version *stream*) *version*)**. This exercise is slightly more efficient than calling **version-control-internals::find-or-make-encached-file-header** for yourself, since it avoids opening the file twice.

Supplying only-section returns a stream that reads text from a single section. You have to supply a version to use only-section.

vci:open-encached-file-stream &key *encached-file* (*file-version* **nil**) *pathname only-section return-boundary-blips* *Function*

Returns a stream that reads the text of a file version from an encached file.

If you supply a pathname instead of an encached-file, this will call **version-control-internals::find-or-make-locked-encached-file** to find the encached file.

You must supply a **version-control-internals::file-version** with file-version.

only-section specifies that the stream only returns the text for the specified section number.

If you specify return-boundary-blips non-**nil**, a **version-control-internals::encached-file-version-stream-blip** is signalled at the beginning of each new section. Your application should **condition-bind** for this condition. Since VC files are stored as lines, this will only be signalled between lines.

vci:open-encached-or-file-stream *vc-pathname* &key *only-section* *Function*

This function is equivalent to calling **open** on vc-pathname, except that you can supply only-section to read a single section.

If the file is encached, this returns a stream that reads the text from the encached copy. Otherwise it returns a stream that reads the text from the VC file in the file system.

vci::vc-file-version *vc-file-stream* *Function*

Returns the **version-control-internals::file-version** for a VC file stream.

vci::vc-file-stream-encached-file-header *vc-file-stream* *Function*

Returns the encached file header for a VC file stream.

vci:encached-file-version-stream-blip *Flavor*

This condition is signalled by **version-control-internals::encached-file-version-stream**s to indicate the transition to a new section. The function **version-control-internals::encached-file-version-stream-blip-blip** returns the **version-control-internals::section-boundary-blip** for the transition. The function **version-control-internals::encached-file-version-stream-blip-stream** returns the stream that signalled the blip, so that handlers can avoid catching each other's blips.

Implementation of VC File Streams

version-control-internals::one-version-file-stream and **version-control-internals::encached-file-version-stream** implement the streams that read from file system files and encached files, respectively.

They are built on some common mixins, which are described here before the streams themselves.

vci::one-version-whole-file-p-mixin *Flavor*

This mixin provides the **:whole-file-p** message for the streams. It has an instance variable initialized with **:force-whole-file**. The functions that create these streams use that keyword depending on whether the stream will return an entire file version or just one section.

vci::diagram-line-blip *Flavor*

This condition flavor is used with the **version-control-internals::return-diagrams-as-lines-mixin** to implement support for the stored editor diagram line protocol. The two streams signal this condition when they encounter an instance as a text record. Then, whoppers on **version-control-internals::return-diagrams-as-lines-mixin** catch the condition and either return or ignore the diagram, depending on whether the application has enabled returning diagram instances.

vci::return-diagrams-as-lines-mixin *Flavor*

This flavor provides whoppers (actually whopper-substs) on the line input stream methods that catch **version-control-internals::diagram-line-blip** conditions. The instance variable return-diagrams-as-lines is **:settable**, since the global protocol for controlling diagram lines is via the **:set-return-diagrams-as-lines** keyword message.

Currently, the whoppers just ignore diagrams when return-diagrams-as-lines is **nil**. They should return the string `<<Diagram line>>` for compatibility with ordinary file streams.

vci:one-version-file-stream *Flavor*

This stream reads text from a VC file in the file system.

The instance variables are as follows:

internal-version — the encached file numeric version for the text version being read.

version — the **version-control-internals::file-version**.

return-vc-pathnames — a flag. When **t**, the **:pathname** message returns a full VC pathname as the stream pathname. When **nil**, it just returns a flat pathname for the VC file.

pathname — the correct VC pathname to return. This will retain **:newest** and related items in the versions, as opposed to the truename.

encached-file-header — just what it says.

parser — the **version-control-internals::parser** instance used to read the file.

stream — the stream open to the VC file.

eof — **t** if the last text has already been returned.

ancestors — an ancestor bitmap for the version being read. This has the same contents, and is used the same way, as the ancestor-bitmap in the **version-control-internals::per-version** structure. See the structure **vci::per-version**.

return-boundary-blips — this is intended to give VC file streams the same ability to signal blips at section transitions as encached file streams have. It is not yet implemented.

only-section — If the application specified that only text from one section should be read, this contains the section number.

only-section-found — a state variable. This starts at **nil**, and is set to **t** if the desired section has been found in the file.

header — the **version-control-internals::parsed-file-header** returned by the parser.

input-buffer-outstanding — used to error-check on **:discard-input-buffer**.

Note that this stream mixes in the standard file stream mixin. This stream provides a **si:pathname** instance variable which it uses for the wholine. This is only important for the file access path which registers streams for display in the wholine. See the section "VC Pathname File Access Path".

The implementation of the stream is a straightforward buffered input stream. Each text record is returned as an input buffer with a `#\Return` appended.

When a diagram line instance is encountered, the **version-control-internals::diagram-line-blip** condition is signalled, leaving it to **version-control-internals::return-diagrams-as-lines-mixin** to deal with them.

vci:encached-file-version-stream *Flavor*

This flavor implements the stream that reads the text of a file version out of an encached file. It is a very simple buffered stream, since all of the hard work happens in the encached file. It has the same somewhat odd set of instance variables for pathnames as **version-control-internals::one-version-file-stream**. For an explanation: See the flavor **vci:one-version-file-stream**.