CVS Training Manual

Table of Contents

1. Introduction to this Manual **1**

2. Introduction to CVS **3**

2.1 What is CVS? **3**
2.2 What does CVS do? **3**
2.3 What is CVS Not? **3**
2.3.1 CVS is not a build system. 3
2.3.2 CVS is not a substitute for management. 4
2.3.3 CVS is not a substitute for developer communication. 4
2.3.4 CVS is not a configuration management system. 4

3. How does CVS compare with other systems? **5**

3.1 CVS vs RCS **5**
3.2 CVS vs SCCS **5**
3.3 CVS vs TeamWare **6**
3.4 Advantages of CVS **6**
3.5 Drawbacks **6**

4. Basic Concepts **7**

4.1 Version Control **7**
4.1.1 Software 7
4.1.2 Documentation 8
4.2 Revision Numbering **8**
4.3 Version Numbering **8**
4.4 RCS Files **9**
4.5 Repository **9**
4.6 Projects **9**
4.7 Modules **9**
4.8 Summary **10**

5. CVS for Software Developers **11**

5.1 Getting Started **11**
5.2 Syntax of CVS commands **11**
5.3 Getting the source code **11**
5.4 Committing your changes **12**
5.5 Cleaning up **12**
5.6 Moving around checked out directories **12**
5.7 Differences **13**
5.8 Logs **13**
5.9 Summary **13**

6. Advanced Concepts **14**

6.1 Adding Files **14**
6.2 Removing Files **14**
6.3 Moving Files **14**
6.4 Updating **15**
6.5 Tags **15**
6.5.1 Tag names 15
6.5.2 How to use a tag 16
6.5.3 How to create a tag 16

6.5.4 Special tags 16
6.6 Branches 16
6.6.1 How to create a branch 17
6.6.2 How to check in on a branch 17
6.7 Merging 17
6.7.1 Merging during an update 18
6.7.2 Merging from a Branch 18
6.7.3 Merging to a Branch 18
6.7.4 Reverse Merging 19
6.7.5 Conflicts 19
6.8 Recursion 20
6.9 Summary 20

7. CVS for Configuration Management Staff 21

7.1 Setting up the Repository 21
7.2 Starting a Project 21
7.3 Creating a Module 22
7.4 Importing Vendor Source Code 22
7.4.1 Importing code from third parties 22
7.4.2 Controlling Local Changes 23
7.4.3 Importing updated versions 23
7.4.4 What is this vendor tag? 24
7.4.5 Generating Patch Files 25
7.5 Software Releases 26
7.5.1 Release strategies 26
7.5.2 Tagging using “rtag” 26
7.5.3 Exporting 27
7.5.4 Branching 27
7.5.5 Magic branch tags 27
7.5.6 Release Summary 27
7.6 Summary 28

8. Software Configuration Management 29

8.1 SCM Policy 29
8.2 Configuration Identification 29
8.3 Configuration Control 30
8.3.1 Change Authorisation 30
8.3.2 Work Authority 30
8.3.3 Change Control 32
8.4 Implementation of Configuration Control 32
8.4.1 Use of the files in CVSROOT 32
8.4.2 Control of commits 36
8.4.3 Control of tagging 36
8.5 Release Procedure 37
8.5.1 Release Authorisation 37
8.5.2 Release Control 37
8.5.3 Version Description Documents 38
8.6 Configuration Status Accounting 38
8.7 Configuration Audits 38
8.7.1 Types of Audits 38
8.7.2 Conducting an Audit 40
8.7.3 Surviving an Audit 40

9. Migration to CVS 42

9.1 Moving from RCS to CVS 42
9.2 Moving from SCCS to CVS 42
9.3 Moving from CVS 1.3 to CVS 1.5 43

10. tkCVS 44

10.1 Introduction 44
10.2 Where do I get tkCVS? 44
10.3 More information on tkCVS 44

11. emacs and CVS 45

11.1 CVS mode 45
11.2 emacs VC 45

12. Remote CVS 46

13. More Information 47

13.1 CVS documentation 47
13.2 CVS books 47
13.3 CVS on the Internet 47
13.4 CVS FAQ 47
13.5 References for CVS 48

14. Acknowledgments 49

1. Introduction to this Manual

This document is intended to be used as a training guide to the CVS system.

It is expected that the presenter of this material will have some experience with CVS, and is presenting the information in this manual to non CVS users. The document has an associated set of overhead slides that can be used in conjunction with the manual.

The expected audience of the training material is as follows:

- Software developers intending to use CVS in their day-to-day development process. These people need to know how to check files in and out of CVS and how the basic mechanics of CVS operate. They do not act as configuration management staff.
- To present material in this manual to this audience, the trainer should:
    - Present sections 2 and 3, briefly in a single session. Do not expect to field any significant level of questions from the audience. This should take approximately 30 minutes.
    - Present section 4 in a little more detail. Expect some questions to be asked on this section.
    - Present section 5, probably in the same session as section 4. Software developers will ask a number of questions at this point. Ensure that all of the concepts in this section are established clearly in the minds of the audience before moving on.
    - During the next session, present section 6. Expect some detailed questions during this session. Do not be overly concerned if the audience do not fully understand all of the concepts in this session -- some of these are best learned by trial and error. This section can also (or later) be used as a "refresher course" for development staff who have previously used CVS and/or previously been exposed to the material in section 5, and so be prepared to re-run this session at a later date. It may be the case in your organisation that only senior members of the development staff (team leaders, etc) need to fully understand all of the details of this section.
    - Do not present sections 7, 8, or 9.
    - In a later session, if your site uses tkCVS (IF NOT WHY NOT?), present section 10. This is best presented to staff who have some familiarity with the basic concepts of CVS. It may also be used as material to present to staff who have no knowledge of CVS, and wish an easy way to use CVS without learning all of the command line concepts.
    - If your site uses emacs, present section 11. I am yet to encounter a CVS site that uses emacs-VC without some modifications, however, and so this may be of limited use to the developers.
    - If your site uses a remote implementation of CVS, present section 12 briefly.
- Configuration Management staff, expecting a deeper understanding of the methods and skills they need to learn in order to maintain a software repository. In some cases these staff are also software developers, although this assumption is not made in this manual.
- To present material in this manual to this audience, the trainer should:
    - Present sections 2, and 3, briefly in a single session. Do not expect to field any significant level of questions from the audience. This should take approximately 30 minutes.
    - Present sections 4 and 5 in a little more detail. Expect some questions to be asked on this section, however the audience attitude may be "we know all of this, keep moving!". Be

flexible in your presentation method on this -- the audience may well have a different idea of what facilities the development staff at their site have or should be allowed access to.

- During the next session, present section 6. Expect some detailed questions during this session. Configuration Management staff should expect to have a fairly detailed knowledge of all of the concepts in this section before you progress any further. Be prepared to break the course at this point, allowing the audience to perhaps examine the CVS man page in more detail, and then to bring back further questions to be fielded at the start of the next session.
- In the next session, present section 7. Be prepared to refer the audience to the appropriate sections of the CVS man page, in particular the man page section on "cvs import" which is covered in this section.
- Note that the CVS man page description of the import function lacks description on how vendor branches actually work, and the use of the vendor branch and release tags. Also, the section on magic branch tags is not covered well in the CVS manual. Be prepared to go into some detail explaining these concepts.
- Present section 8. This section deals mostly with the concepts of software configuration management and deals somewhat with methods to implement these concepts. It may take some tailoring to the audience that you are presenting it to, otherwise present these as general concepts that are applicable across organisations, rather than specific rules that need to be applied in any particular organisation. Refer the audience to specific information on the implementation of their own CM requirements (MIL-STD-973 may be a starting point for an audience in the defence industry).
- Present some or all of section 9 as appropriate. This should be relatively quick depending on whether they have existing SCCS or RCS repositories that need to be converted to CVS. There are of course plenty of consulting opportunities in this area.
- In a later session, if your site uses tkCVS (IF NOT WHY NOT?), present section 10. This is best presented to staff who have some familiarity with the basic concepts of CVS. It may also be used as material to present to staff who have no knowledge of CVS, and wish an easy way to use CVS without learning all of the command line concepts.
- If your site uses emacs, present section 11. I am yet to encounter a CVS site that uses emacs-VC without some modifications, however, and so this may be of limited use to the developers.
- If your site uses a remote implementation of CVS, present section 12. Sections 9 through 12 may be presented in one session if appropriate.
- Section 13 contains some references that the audience may find useful. If appropriate, deal with section 13 as a hand out, rather than presenting it as tutorial material.

This document is not any of the following:

- A CVS reference manual. The correct CVS reference manual is the **`cvs`** man page distributed with the CVS system. If you have CVS installed on your system, typing **`man cvs`** at the shell prompt will give you the information that you require.
- A complete CVS users' guide. A useful CVS users' guide has been written by Per Cederqvist of Signum Support AB (Sweden), and this document is also distributed with the CVS source code. The document is in the doc/ directory in GNU texinfo format as well as printable PostScript format.

2. Introduction to CVS

2.1 What is CVS?

"CVS" is an acronym for the "Concurrent Versions System".

CVS is a "Source Control" or "Revision Control" tool designed to keep track of source changes made by groups of developers working on the same files, allowing them to stay in sync with each other as each individual chooses.

2.2 What does CVS do?

CVS is used to keep track of collections of files in a shared directory called "The Repository". Each collection of files can be given a "module" name, which is used to "checkout" that collection.

After checkout, files can be modified (using your favourite editor), "committed" back into the Repository and compared against earlier revisions. Collections of files can be "tagged" with a symbolic name for later retrieval.

You can add new files, remove files you no longer want, ask for information about sets of files in three different ways, produce patch "diffs" from a base revision and merge the committed changes of other developers into your working files.

2.3 What is CVS Not?

2.3.1 CVS is not a build system.

Though the structure of your Repository and modules file interact with your build system (e.g. a tree of Makefiles), they are essentially independent.

CVS does not dictate how you build anything. It merely stores files for retrieval in a tree structure you devise.

CVS does not dictate how to use disk space in the checked out working directories. If you require your Makefiles or build procedures to know the relative positions of everything else, you wind up requiring the entire Repository to be checked out. That's simply bad planning.

If you modularize your work, and construct a build system that will share files (via links, mounts, VPATH in Makefiles, etc.), you can arrange your disk usage however you like.

You have to remember that any such system is a lot of work to construct and maintain. CVS does not address the issues involved. You must use your brain and a collection of other tools to provide a build scheme to match your plans.

Of course, you should use CVS to maintain the tools created to support such a build system (scripts, Makefiles, etc).

2.3.2 CVS is not a substitute for management.

You and your project leaders are expected to plan what you are doing. Everyone involved must be aware of schedules, merge points, branch names, release dates and the range of procedures needed to build products. (If you produce it and someone else uses it, it is a product.) CVS can't cover for a failure to manage your project.

CVS is an instrument for making sources dance to your tune, but you are the piper and the composer. No instrument plays itself or writes its own music.

2.3.3 CVS is not a substitute for developer communication.

When faced with conflicts within a single file, most developers manage to resolve them without too much effort. A more general definition of "conflict" includes problems too difficult to solve without communication between developers.

CVS cannot determine when simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. Its concept of a "conflict" is purely textual, arising when two changes to the same base file are near enough to spook the merge command into dropping conflict markers into the merged file.

CVS is not capable of figuring out distributed conflicts in program logic. For example, if you change the arguments to function X defined in file A and, at the same time, edit file B, adding new calls to function X using the old arguments, you are outside the realm of CVS's competence.

Acquire the habit of reading specs and talking to your peers.

2.3.4 CVS is not a configuration management system.

CVS is a source control system. The phrase "configuration management" is a marketing term, not an industry-recognised set of functions.

A true "configuration management system" would contain elements of the following:

- Source control.
- Dependency tracking.
- Build systems (i.e. What to build and how to find things during a build. What is shared? What is local?)
- Bug tracking.
- Automated Testing procedures.
- Release Engineering documentation and procedures.
- Tape Construction.
- Customer Installation.
- A way for users to run different releases of the same software on the same host at the same time.

CVS provides only the first.

3. How does CVS compare with other systems?

3.1 CVS vs RCS

CVS uses RCS to do much of its work and absolutely all the work of changing the underlying RCS files in the Repository.

RCS comprises a set of programs designed to keep track of changes to individual files. Of course, it also allows you to refer to multiple files on the command line, but they are handled by iterating over individual files. There is no pretence of coordinated interaction among groups of files.

CVS's main intent is to provide a set of grouping functions that allow you to treat a collection of RCS files as a single object. Of course, CVS also has to do a lot of iteration, but it tries its best to hide that it is doing so. In addition, CVS has some truly group-oriented facets, such as the modules file and the CVS administrative files that refer to a whole directory or module.

One group aspect that can be a bit confusing is that a CVS branch is not the same as an RCS branch. To support a CVS branch, CVS uses “tags” (what RCS calls “symbols”) and some local state, in addition to RCS branches.

Other features offered by CVS that are not supported directly by RCS are

- Automatic determination of the state of a file, (e.g. modified, up-to-date with the Repository, already tagged with the same string, etc.) which helps in limiting the amount of displayed text you have to wade through to figure out what changed and what to do next.
- A copy-modify-merge scheme that avoids locking the files and allows simultaneous development on a single file.
- Serialisation of commits. CVS requires you to merge all changes committed (via "update") since you checked out your working copy of the file. Although it is still possible to commit a file filled with old data, it is less likely than when using raw RCS.
- Relatively easy merging of releases from external vendors.

Migration from RCS to CVS is relatively simple -- RCS files are directly supported as a part of the CVS repository. Most of the effort here is taken up in structuring the repository, adding new files required by CVS (modules, loginfo, etc), and physically moving the RCS files into the new CVS repository.

3.2 CVS vs SCCS

SCCS is much closer to RCS than to CVS, so some of the previous section applies.

You might want to take a look at Walter Tichy's papers on RCS, which are referred to in the RCS man pages.

Note that SCCS only supports text files -- CVS and RCS also support arbitrary (binary) files.

Migration from SCCS to CVS is a two step process. Firstly, CVS provides a script called **`sccs2rcs`** that will convert all of your **`"s."`** SCCS files to RCS **`",v"`** files. Secondly, the **`",v"`** revision files must be moved in to the CVS repository.

3.3 CVS vs TeamWare

TeamWare is a configuration management tool from Sun Microsystems. It uses the same copy and merge model as CVS. The central abstraction is a workspace, which corresponds to either a CVS branch or a checked out module. TeamWare allows you to manipulate workspaces directly, including moving and merging code between workspaces. You can put your workspace on tape and continue to work with it at home, just like you can with CVS. TeamWare is built upon and compatible with SCCS.

All TeamWare tools are graphical. The CodeManager tool will display the project as a tree of workspaces, and allows you to manipulate them with drag and drop. The other tools are VersionTool that displays and manipulates a revision history of a single file, CheckPoint that will create symbolic tags, MakeTool, a make compatible tool with a GUI, and FileMerge which will interactively merge files when needed (like emerge for emacs). If you have a sun, you can try **`/usr/old/filemerge`** for an old SunView version of FileMerge.

Email: sunprosig@sun.com

3.4 Advantages of CVS

Of the commercial and non-commercial source code control products, CVS is an obvious choice: it's free software (GNU), it supports many platforms, it comes with all the source, and it has ample support via mailing lists or Usenet newsgroups. Also, its versions database engine is vanilla RCS, which has been around for some time and simply works ok. Incidentally, since RCS revision files are simply ASCII and RCS operation is pretty straightforward, manual troubleshooting is easy. This is a thing to consider, even if a proper backup strategy is being employed.

CVS adds the notion of “modules”, or hierarchical collections of files, to the flat filespace that RCS handles. CVS operation is not limited to single files but can actually refer to a module in its entity.

There is no locking mechanism in CVS. Developers work concurrently in private areas (sandboxes) and merge their work with that of co-workers on demand. We consider this a pretty neat feature of CVS, but it requires overhead in terms of communication between project teams and some procedure for painlessly integrating changes into the source base. This is hard anyway. I doubt that there are tools which simply do it for you, so the CVS way of doing things is probably not worse than locking access to files.

3.5 Drawbacks

- CVS does not handle non-text (i.e. binary) files gracefully (the problem here is how to diff revisions meaningfully and how to store binary data in deltas as opposed to simply dumping the file).
- CVS has a somewhat peculiar way of doing some things. See FAQ for examples. CVS is not hard to use, but it's very helpful to understand the underlying ideas.

4. Basic Concepts

4.1 Version Control

Version control is a management exercise. The objective behind version control is to enable a particular "snapshot" of a development process to be recovered at some stage in the future (i.e. after the development has moved on from the snapshot time).

Without version control, a project can encounter the following difficulties:

- A software release is made to a customer. Some time later a new software release is made, theoretically providing some enhancements or fixing some bugs. The customer complains that the new release has one particular bug that prevents it from being useable, and wants to "roll-back" to the previous release until this bug is fixed. Without version control, there is no roll-back mechanism.
- A software developer is making a change to a source code file. During the process of that change, a serious mistake is made, and the developer needs to recover the file to the point before the change was made. Without version control there is no recovery mechanism.
- A new software release is made to a customer. The customer wishes to be given a list of changes that have been made to the software since the previous release. No change control mechanism is available without version control.

Version control can be implemented in a number of ways. The two most common methods are:

- Each time a software release or software change is made, a backup copy (on tape or some other physical medium) is made. Tape copies of previous releases of software are stored in a software library, or archival storage. This method is known as "Physical Version Control".
- A mechanism of storing multiple different revisions of the same file within the file system is established. This can be done either at the operating system level (by making changes to the file system code, or by implementing an operating system that supports multiple revisions of files, such as VMS); or by implementing an additional piece of software that enables "recovery" of an old revision of a file. Software of this type is known as "version control software".

4.1.1 Software

Version control of software source code files is the most obvious use for version control software. Software source code is typically stored in ASCII text files which are small, and relatively inexpensive to store multiple versions of.

Modern version control software (such as CVS, RCS, and SCCS) do not store a complete copy of each revision of each file. Instead, they store one "master" revision, and then each other revision in the form of a set of "differences" between that revision and the master revision.

Standard file comparison software such as Unix or GNU **`diff`** is used to determine the differences between any two revisions of a software source file.

Because programmers tend to program on a "line by line" basis, adding new variables and program commands by adding complete new lines of source code into the program file, this is a valid and useful method of storing software source code.

4.1.2 Documentation

Version control of documentation files (including such things as word processing documents, spreadsheets, diagrams, etc) is more difficult. This is because such documents do not tend to be stored in ASCII text file format.

Standard "diff" utilities break a file into lines based on an ASCII end-of-line character, and compare two files by comparing the complete contents of each line in the file. The position of ASCII end-of-line characters in a binary file, however, is somewhat arbitrary, and this does not usually lead to a good comparison method.

For this reason, basic version control software such as SCCS will not handle non-text files.

Very few documentation systems have in-build version control software. None of the PC based word processors do this. Therefore, for such files, CVS enables version control of the files on an arbitrary line-by-line basis. This does not usually lead to an efficient storage method, but it is better than tapes and printouts, and in the worst case uses only a few tens of bytes more than storing each revision individually.

4.2 Revision Numbering

Revision numbering is what sets two different revisions of the same file (or project) apart.

Typically, revision numbering is used by organisations such as MicroSoft to indicate how many bugs they have in their software. Microsoft Word version 6 has 3 times as many bugs as Word version 2. Windows 95 therefore will have 32 times more bugs than Windows 3.

Revision numbering is used by CVS and RCS to mark revisions of a file in an RCS file. An RCS revision number always has an even number of decimal integers, separated by dots. For example, "1.1", "1.2" and "1.2.2.2" are valid RCS revision numbers.

CVS is not limited to development along a single stream. The CVS revision tree can be split into branches, where each branch is a complete line of development. Changes made on one branch can then be moved back on to the main trunk.

Each branch has a branch number that is similar to a revision number but has an odd number of integers. For example, revision 1.2 can have a branch with a branch number of 1.2.2. The first revision on this branch will be revision number 1.2.2.1.

4.3 Version Numbering

I almost never use the term "version number". This is because it has two possible meanings. They are:

- Version number of a file, as in myfile.c version 1.2.
- Version number of a software product, such as "Brewery Control Software version 1.2a".

Instead of referring to version numbers of files, I refer to "revision numbers". Similarly, I use the term "release numbers" to refer to version numbers of a software product.

CVS applies some rules and methods to how you apply revision numbers to a file. CVS does not apply any rules as to how you number your software releases.

4.4 RCS Files

An RCS file is a file containing multiple revisions of a single source file that is managed by RCS.

RCS is the basic backbone of CVS, in that it does all of the work relating to these revision controlled files, including adding new revisions, getting revisions out of the file, reading and interpreting the contents of the file, and applying tags to the file.

4.5 Repository

CVS stores all of its RCS files in a central location called a repository. A repository is a directory stored on a file server with a name known to all developers (for example, /usr/local/cvsroot).

The repository is normally NFS mounted to all workstations that are running CVS. CVS 1.5 and later releases have the facility to operate remotely over a TCP/IP network using the rsh protocol, and so NFS mounts are no longer strictly necessary.

Developers do not access or modify any of the files in the CVS repository. The normal process of working with the repository is to obtain a local working copy of a part of the repository (normally a module), make local changes, and merge these changes back in to the repository. This style of source control model is known as “copy-modify-merge”.

4.6 Projects

A unit of software that can fit into a single top level hierarchy can be called a “project”. Some organisations may require a separate repository for each project, some may have a project spread over a number of repositories (I personally think that this is a bad idea), and some have a number of projects contained in one repository. Some organisations only ever have one project.

It is up to you to decide what constitutes a project in your organisation. I do not attempt to describe project management, only source code management. There are a lot of good books written about software project management (and no doubt some really bad ones).

Consider, however, the case of four separate and distinct software products produced by a company. Each software product is produced by a separate team of software engineers, and each team works independently of each other. I would describe each product as a project in this case.

Extend the above case to a situation where the above products all share one common utilities library. This utilities library contains some graphics routines, database management routines, etc, that are in use by all four products. There may be a separate team working on the utilities library, or it may be the case where some engineers from all four teams periodically update the utilities library as required. In this case I would describe each product as a project, and the utilities library as an additional project.

The point behind this explanation is that CVS can handle multiple projects in a repository. Each project should in most cases have its own top level directory, and their should be a hierarchy of directories and modules under the top level directory for the project.

4.7 Modules

A module is a discrete part of a project, or in some cases the whole project itself.

A module is best kept to a unit that a single programmer can be made responsible for. CVS's check-in and check-out procedures all work on a module by module basis, and so a software engineer will typically check out a module, make some changes, and then check in those changes.

A module can have multiple files, this is fairly common.

A module can have subdirectories, although this is less common. If the majority of your modules are large and each of them have several subdirectories, you should consider re-constructing your development environment so that each subdirectory is a module.

In many cases, a module will have a Makefile so that it can be compiled, and possibly linked into an executable or an archive library, separately. If a module depends significantly on other modules for compilation, then it is perhaps too small.

Modules are listed in the CVS modules file -- this is a special file contained in the CVS repository in the CVSROOT directory.

4.8 Summary

This section presented the overall concepts of software version control. You should have a basic understanding of these concepts before you attempt to learn any of the CVS functionality.

5. CVS for Software Developers

In this section I will present a sample CVS session. I will assume that a CVS repository has been established.

5.1 Getting Started

To use CVS, you need to have a CVS repository. You should ask your CM staff for the location of the repository.

Once you know the location of the repository, you need to point your CVSROOT environment variable at the repository. Do this using the shell commands export or setenv (depending on what sort of shell you are using). You should set this in your .cshrc, .profile or .env file.

5.2 Syntax of CVS commands

All CVS commands follow the same general syntax. This is:

```
cvs <global_options> command_name <command_options> [<file> ...]
```

- <global_options> can be used to define the way that the CVS command generally operates. Global options are those options that are available for all commands. Some common global options are:

-q	Causes the command to be quiet.
-Q	Causes the command to be very quiet.
-H	Displays some help information about the command, but doesn't actually execute the command.
-n	Do not execute the command, just show what would be done by executing the command. This will generate reports, etc, but will not actually change the repository.

- <local_options> are also used to define the way that the CVS command works, however local options are specific to the command being run. Some common local options are:

-r <rev>	Operate on the revision <rev> rather than the head revision.
-l	Local -- do not recurse through subdirectories.
-f	When you checkout by a tag, and the tag does not exist on a particular file, check out the head version instead. Normally, the file is not checked out.

A full list of CVS commands and all of their options is contained in the **`cvs`** man page.

5.3 Getting the source code

To get source code from CVS, you check out the code with the following command:

```
cvs co <module>
```

This checks out the head revision of the module called **<module>**. Instead of checking out the head revision, you may check out a particular revision of the software with the following command:

```
cvs co -r <revision> <module>
```

cvs co will create a new directory for the module you have checked out. This directory will have the same name as the module name.

After the initial checkout you are free to hack away at the files you have checked out.

5.4 Committing your changes

To commit your changes to cvs, you can check in the code with the following command:

```
cvs ci
```

This will do the following things:

- Check the files in the directory that you are in to see if there are any modifications.
- Asks for a revision log comment for the files you are checking in.
- For each modified file:
  - Check to ensure that the file has not changed in the repository since you checked it out.
  - If it has not changed, updates the repository with your working file. This "commits" your change to the repository.

If the files that you have modified have also been modified in the repository, then you have a "conflict". This conflict must be resolved by updating the files before you can check in.

5.5 Cleaning up

After you have checked in your files, you are responsible for cleaning up the directory you have created. You can safely remove a directory that has been checked out of CVS using **rm**.

Some sites may require a "cvs release" operation instead of rm. The syntax for this is:

```
cvs release -d <module>
```

5.6 Moving around checked out directories

There are a few important points to note about directories that you check out of the repository:

- When you check out a module, this creates a "module" directory.
- When you check in, diff, or log a module, do it from within the module directory.
- When you release a module, do it from the directory above the module directory.

For example, if you check out the "gruff" module, edit it, check it in, and release it, you must follow this sequence:

```
cvs co gruff
cd gruff
[vi ... emacs ... textedit ... nedit ... etc]
[cvs diff; cvs log; cvs status ]
```

```
cvs ci
cd ..
cvs release –d gruff
```

5.7 Differences

While working in a module directory you can get a difference listing for the files in the module.

You can see the difference listings within an entire directory using the following command:

```
cvs diff
```

You can see the difference listings for a file or group of files using the following command:

```
cvs diff [<file> ...]
```

5.8 Logs

A cvs log (or revision log) is a list of revisions that have been created for a file, and a comment for each of those revisions. You create a revision comment each time that you commit a file.

To view a cvs log, type the following command:

```
cvs log [<file> ...]
```

5.9 Summary

This section covered some of the basic CVS commands. After attending this section, software engineers should be able to operate the most basic CVS functions, including checking files out of the repository, and committing those changes to the repository.

These are the most fundamental elements of CVS. Ensure that you have understood the commands presented in this section before proceeding further with the training.

6. Advanced Concepts

This section tackles some of the more complex concepts and tasks that associated with CVS.

6.1 Adding Files

You can add a file to the cvs repository relatively easily. Just follow these steps:

- Check out the module which you want to add the file to.
- Create the file in the module directory. You cannot add a file to the repository unless you have created it first! You can use your favourite text editor for this.
- Run the command **`cvs add file`**. This will "mark" the file for addition to the repository. Note that the file is not in the repository yet -- like all changes you must commit it first.
- **`cvs commit file`**. This will commit your change to the file -- in this case the change you have made is to add the file.

6.2 Removing Files

Files are removed from the repository in the same way in which they are added:

- Check out the module which you want to add the file to.
- Delete the file from the module directory. You cannot delete a file from the repository unless you have deleted it in your working directory it first.
- Run the command **`cvs remove file`**. This will "mark" the file for deletion from the repository. Note that the file is still in the repository -- like all changes you must commit it first.
- **`cvs commit file`**. This will commit your change to the file -- in this case the change you have made is to remove the file.

Note that files that you remove from the repository do not actually dissappear into hyperspace -- they are still stored in the repository in a special location called the "Attic". If the file has tags, and you check out the file using a tag name, it will re-appear.

Files that have been deleted from the repository (i.e. are in the Attic) do not have a "head" version any more. In other words, the latest change to that file is that the file has been removed. This means that if you check out a module using the head version (cvs co), that file will not appear. Do not be overly concerned about seeing files that you have deleted from the repository hanging about in Attic directories.

6.3 Moving Files

The best way to move a file in the repository is to delete it from one place (cvs remove) and add it in the new place (cvs add).

If you want more information about the complexities of moving files in the repository, check the FAQ. Approach any method of moving files other than remove + add with extreme caution.

6.4 Updating

CVS uses a copy-modify-merge model of software version control. That is, when you check a file out of the repository using “cvs co”, that does not stop other developers also checking out the same file.

When you attempt to commit changes that you have made to a file, it may be the case that another developer has also committed changes to that file since you last checked it out of the repository. This means that you have to update your local copy of the file before you can commit your changes.

The command to perform an update is:

```
cvs update
```

Note that you do an update from the module directory that will have been created by cvs co.

cvs update produces some terse but informative output when it runs. For each file, it produces one of the following types of output:

`U file`	The file has been updated from the repository. Your local copy has been changed to match a later revision that has been committed by another developer.
`M file`	Your file has been updated to include changes from the repository. A successful merge has occurred.
`C file`	There is a conflict -- a failed merge has occurred.

If you have a conflict, you need to resolve it and perform another update before you can commit your changes.

6.5 Tags

CVS tags are used within a CVS repository to mark revisions of particular interest to the development process.

A tag marks a version of a file in a CVS module with an alphanumeric string. Different versions of different files may be tagged with the same tag to indicate that they are at an equivalent stage in development, or belong to a similar environment.

When files are released from a CVS repository to a customer, the files are usually released using a tag name.

6.5.1 Tag names

Tag names can be anything the software engineers or SCM staff can make some sense out of. Here are some restrictions on tag names:

- Tag names must not start with a number. For example, 1_5_alpha is not a valid tag name.
- Tag names must not contain dots (so they do not become confused with revision numbers). For example, release-1.5 is not a valid tag name, but release-1_5 is valid.

6.5.2 How to use a tag

A tag may be used in any place that a revision number is used in. For example, the following command checks out from the repository a module called myfiles, with revision number 1.2 (i.e it checks out revision 1.2 of each of the files

```
cvs co -r 1.2 myfiles
```

Assuming that there is a tag called "release" applied to one of the revisions of this file, the following command is used to check out the revision with the "release" tag:

```
cvs co -r release myfiles
```

The "release" tag need not have been applied to the same revision of each file.

6.5.3 How to create a tag

Tags can be created on files using the "cvs tag" or "cvs rtag" commands. The syntax of the cvs tag command is:

```
cvs tag [options] tag_name [<file> ...]
```

You can use the "cvs tag" command on a file or module that you have checked out of the repository. For example, the following command will tag the revision of myfile.c that you have checked out of the repository with the tag name "release":

```
cvs tag release myfile.c
```

If you want to tag a particular version of a file, you must first acquire that version from the repository, then tag it, as follows:

```
cvs update -r 1.4 myfile.c
cvs tag release myfile.c
```

Normally the purpose of applying a tag is to mark a special version of a file to be used in a particular release.

6.5.4 Special tags

There are two special tags that apply to each checked out file:

BASE	This is the tag belonging to the revision of the file that has actually been checked out of the repository. For example, if you check out revision 1.4 of a file, then the repository is updated to revision 1.5, BASE will refer to 1.4 (until you update your local copy to 1.5).
HEAD	The head of the branch that you have checked out -- in most cases the most recent trunk version of the file in the repository. In the above example, the HEAD revision would be 1.5.

6.6 Branches

Branches allow the development of different lines of software at the same time.

The most common use of branches is to allow software developers to make changes to a previous version of software while still allowing development of a new version. In this case, the previous version

will be tagged when it is released, and then a second tag, called a magic branch tag, will be applied to the software at the same location as the first tag.

After applying the branch tag, a check-out can be made on either the branch or the main revision of the software.

6.6.1 How to create a branch

A branch is created when you apply a branch tag. Branch tags are applied with the "-b" option. For example:

```
cvs tag -b branch_2_0 myfile.c
```

Branch tags are commonly applied to a previous release of a software module. This can be achieved by updating to that relese before creating the tag. For example:

```
cvs update -r release_2_0 myfile.c
cvs tag -b branch_2_0 myfile.c
```

In this example, **`old_version_tag`** could be a tag (for example **`release_2_1`**) or a revision number (for example, **`2.1`**). Note that **`cvs tag`** does not support the **`-r`** option.

Alternatively, branch tags can be applied to an entire module using **`cvs rtag`**. For example:

```
cvs rtag -r release_2_0 -b branch_2_0 gruff
```

Note that in this case the **`-r`** option is supported.

6.6.2 How to check in on a branch

Checking in on a branch first requires that you check out on that branch. You can check out a file on a branch by using cvs co or cvs update, with the "-r" option to indicate what branch tag you need.

A sample session, checking out a module that has been branched and checking back in on the branch, looks a bit like this:

```
cvs co -r branch_tag gruff
cd gruff
[vi ... emacs ... textedit ... nedit ...]
cvs ci
```

If you already have a module that has been checked out, and you wish to work on a branched revision of one file in that module, you need to use the "cvs update" command to get that branched revision.

```
cvs co gruff
cd gruff
cvs update -r branch_tag gruff_main.c
[vi ... emacs ... textedit ... nedit ...]
cvs ci gruff_main.c
```

6.7 Merging

Merging is the process of taking two modifications to a revision of a file, and turning them into one modification. This modification is then applied to the original revision to create a new revision.

6.7.1 Merging during an update

Normally, a merge will occur when required if you do a cvs update. When a merge occurs as a part of a cvs update, one of two possibilities will occur:

`M file` Your file has been updated to include changes from the repository. This can happen when you change a file, and that file is also changed in the repository, but the two changes are compatible. The file in your directory will now look like you have made your changes to the new version of the file, rather than the old version.

`C file` There is a conflict. This can happen when you change a file, and that file is also changed in the repository, but the two changes are not compatible. The file in your directory will now have conflict markers in it.

If you have a conflict, you need to resolve it and perform another update before you can commit your changes.

6.7.2 Merging from a Branch

You can merge changes that have been committed to a branch back to the head version using CVS update. To do this, use the following procedure:

- Check out the head version of the file you want updated using **`cvs co`** or **`cvs update`**.
- Run this command: **`cvs update -j <branch-tag> file`**. This will merge the changes from the branch tagged with <branch-tag> into your working version. This will produce one of two results:
  - **`M file`** -- This means that the merge has happened successfully.
  - **`C file`** -- This means that there is a conflict merging the file.
- If there is a conflict, resolve it, and do **`cvs update file`** one more time in order to make sure your version can really be applied to the head.
- **`cvs ci`** your version of the file. You have now created a new version on the head of the file that contains the latest head version of that file, complete with any changes to that file that were made on the branch.

6.7.3 Merging to a Branch

The process of merging to a branch is similar to the process of merging from a branch, except in reverse. This time you will need to indicate what part of the main trunk you want merged to the branch by using two -j options.

- Check out the branched version of the file you want updated using **`cvs co -r branch`** or **`cvs update -r branch`**.
- Run this command: **`cvs update -j <rev1> -j <rev2> file`**. This will merge the changes made between revisions rev1 and rev2 back to your working copy of the file. Note that you now have a version of the file that you should be able to apply back to the branch. This will produce one of two results:
  - **`M file`** -- This means that the merge has happened successfully.

  - **`C file`** -- This means that there is a conflict merging the file.

- If there is a conflict, resolve it, and do **`cvs update file`** one more time in order to make sure your version can really be applied to the end of the branch.
- **`cvs ci`** your version of the file. You have now created a new version on the end of the branch of the file that contains the latest branched version of that file, complete with any changes to that file that were made between the two versions (ver1 and ver2) that you specified.

6.7.4 Reverse Merging

Reverse merging can be useful in some circumstances -- when you want to remove changes that were made to the main trunk of a file between two revisions (rev1 and rev2) of a file so that the file can be committed back to a branch without those changes.

Here is how to do a reverse merge:

- Check out the branched version of the file you want updated using **`cvs co -r branch`** or **`cvs update -r branch`**.
- Run this command: **`cvs update -j <rev2> -j <rev1> file`**. Note the order of the revisions -- rev1 is the first revision, rev2 is the second revision, but they are specified in the opposite order on the command line. This will remove the changes made between revisions rev1 and rev2 from your working copy of the file. Note that you now have a version of the file that you should be able to apply back to the branch. This will produce one of two results:
  - **`M file`** -- This means that the merge has happened successfully.
  - **`C file`** -- This means that there is a conflict merging the file.
- If there is a conflict, resolve it, and do **`cvs update file`** one more time in order to make sure your version can really be applied to the end of the branch.
- **`cvs ci`** your version of the file. You have now created a new version on the end of the branch of the file that contains the latest branched version of that file, but has had removed any changes to that file that were made between the two versions (ver1 and ver2) that you specified.

6.7.5 Conflicts

To resolve a conflict that occurs during a merge, you must do the following:

- Edit the file.
- Find any conflict markers that occur in the file. These will be lines beginning with either “>>>”, “<<<”, or “===”. The section after the first such conflict marker will indicate what was in your local copy of the file. The section after the second such conflict marker will indicate what was in the repository (HEAD) version of the file. The section after the third marker will indicate what was in the original (BASE) version of the file.
- You need to select one of the three sections from each conflict area, and leave this in the file. Delete the other two sections.
- Delete all of the conflict markers.
- Do a “cvs update” on the file again to indicate to CVS that your file is ready to be committed. This may perform another merge, and generate new conflict markers, which you will have to resolve.

Once you have resolved the conflicts in the file you are ready to commit that file.

CVS will not allow you to commit a file that has unresolved conflicts in it.

6.8 Recursion

See the section titled “recursion” for more information on this topic. That’s a joke, son.

Most CVS commands are recursive. That is, when you execute the CVS command in a checked out directory, and you have subdirectories in that directory, the command will enter each of those subdirectories in turn and attempt to execute that same command in that subdirectory.

Sometimes, CVS commands are not recursive. CVS commands are not recursive when:

- You specify a file name to the cvs command. For example, “cvs ci myfile.c” will only commit the file called myfile.c, and not recurse in to the subdirectories.
- You specify the “-l” option to the CVS command. For example, “cvs ci -l” will only commit the changes in the local working directory, not in any subdirectories.

6.9 Summary

This section has covered some of the more complex issues of CVS. Do not be overly concerned if you do not fully understand all of these concepts -- some of them are best learned by trial and error (but do this on scratch files not on the real copies of the files in your repository!).

The CVS manual is a good source of reference for these commands. If you have any difficulty with these commands you should read the manual first. There is also a CVS users’ guide that walks through some of these issues in a little more depth than the manual.

7. CVS for Configuration Management Staff

This section covers some details regarding the establishment and maintenance of a CVS repository. Normally it is only Software Configuration Management staff (or "CVS administrators") that will perform these tasks.

I assume by this stage that you have a reasonable understanding of the basic concepts and operation of CVS. You should, as you progress through this section, read and attempt to gain an understanding of the CVS documentation. This includes the CVS manual page and the CVS users guide.

7.1 Setting up the Repository

There are several issues involved in setting up a CVS repository. Here are a few:

- Allocate enough disk space to hold the data in the repository. Allow for growth in the repository as new revisions of files are created and checked in.
- CVS can be run in either client-server mode or using a local repository. The local repository can be mounted to all machines on which CVS is needed using NFS -- this is probably more satisfactory in a local area network environment. For a wide area network where data link speeds are slower, using a remote repository can help save some bandwidth.
- Create the repository using the **`cvsinit`** script supplied with CVS -- this usually does a pretty good job of setting up a new repository.
- Plan your repository before you begin importing files. Separate your files into modules and group these modules in directories. A repository structure that closely mirrors the structure of the development environment is a good start, although you may think more closely about how your development environment is structured -- are the files that normally sit in one directory really related?
- Make sure that your development environment can be completely reconstructed from your CVS repository -- preferably using one high level export or checkout operation. This means that files such as Makefiles, installation and build scripts, etc, all have to be included in the repository.

After the repository is created, you can start importing files into it.

7.2 Starting a Project

It is best to have one top level directory in your CVS repository for each project. It is not always advisable to have multiple repositories in any organisation (even one per project) because software engineers can become confused when moving from one project to another (for example, scripts that explicitly set CVSROOT do not work any more when you change repositories).

To start a project from scratch within the repository, it is simply a matter of creating new directories within the repository and populating them with RCS files. There are several approaches to this:

- If you already have RCS files, these can be copied directly in to the repository. This process is relatively simple, and will be described later.
- If you have SCCS files, these can be converted to RCS files and copied into the repository. This will preserve all of your existing version history. Conversion from SCCS to CVS is covered in detail in a later section.

- If you just have plain ASCII files with no previous version history, consider using the **`cvs import`** command to import them into the repository. This has the advantage of being able to import an entire tree of raw text files into a proper version controlled heirarchy. **`cvs import`** was originally designed to import vendor sources but can be equally well used to import legacy source code into CVS -- just ignore the vendor branches that **cvs import** creates.
- If you have no files to start with, just leave the directories empty. The software engineers will in time populate these with files using cvs add.

Note that in the case where you do not have any files, there is no need to create any empty directories either -- this is just for convenience because a directory can be created in the repository just as easily as a file (using **`mkdir`** and **`cvs add directory-name`**). It may pay to have SCM staff create these directories, however, so that everyone is working from directories with the same name!

7.3 Creating a Module

Creating a module in a CVS repository is usually done at the time the module is imported or added to the repository.

When you add or import a new directory into the repository, you should ensure that you add an entry to the modules file. This is done manually by checking out the modules file, editing it, and checking it in again.

7.4 Importing Vendor Source Code

There are some considerations to be made when you are accepting vendor source code (i.e. anything that has not been developed by your own organisation) into your CVS repository. These include:

- Consider making the vendor source code a separate project, unless it is specifically part of a larger project.
- Make sure that the vendor code is easily identifiable to ensure that it can be kept separate from what you have in your own development stream.
- Beware of vendor source code that contains multiple streams or branches.

7.4.1 Importing code from third parties

The **`cvs import`** command can be used to import entire heirarchies of source code into the repository, and it is the best command to use when importing code from third parties. The syntax of the command is:

```
cvs import [options] <module> <vendor-tag> <release-tag>
```

Three arguments are required to this command, these are:

`module` — The location in the repository where the source code will be stored. Note that in this case module refers to an absolute location in the repository, not a module name as referred to in the modules file.

`vendor-tag` — The source code (if this is a first time import) will be stored on a special branch called the VENDOR branch. This is always branch number 1.1.1. This vendor tag will be used as a branch tag for the 1.1.1 branch.

release-tag	The first revision of the software on the 1.1.1 branch will be revision 1.1.1.1. This release-tag will be applied to the 1.1.1.1 version as it is created in the repository. This tag can then be used as the base point of comparisons between initial vendor versions and subsequent version, or used as an argument to **cvs co** in the same way as any other tag name.

I suggest the following strategy for naming release and vendor tags:

- The vendor tag is just used as a branch tag -- this is less important than the release tag, because no checkouts will ever be done explicitly on this tag. I use something that may have some relevance to the software vendor in question -- for example the vendor name or some contraction of it. If you don't want to use the vendor name (or in the absence of a reasonably obvious choice), I just use the word "VENDOR".
- The release tag should mean something in terms of the software release number. I often use something based on the release number of the software (remember that a tag name cannot have dots in it), for example release-1-4. In the absence of a sensible release number, I use the word "baseline-n" and just let n start at 1 and go up from there.

7.4.2 Controlling Local Changes

Local developer changes to third party code are easily distinguishable in that they will not appear on the vendor branch. Vendor branch changes are only ever created by the **cvs import** command.

For example, assume a file, myfile.c, has just been created in the repository. The revision of myfile.c that is checked out using a **cvs co** command will be the head of the vendor branch (1.1.1.1). If this file is then modified and committed, the revision created will be placed on the head of the trunk (1.2):

```
cvs co gruff
cd gruff
cvs status myfile.c                    # revision 1.1.1.1 checked out
[vi ... emacs ... nedit ...]
cvs ci myfile.c                        # revision 1.2 created.
```

7.4.3 Importing updated versions

Importing new versions of vendor supplied code is as simple as repeating the import procedure. The new source code will be added to the vendor branch. The command to import new versions of vendor code is the same import command as used previously:

```
cvs import <module> <vendor-tag> <release-tag>
```

The <module> and <vendor-tag> arguments will be the same as before. The and release-tag argument should be adjusted to reflect the new version of software being imported (for example, the release-tag should indicate which new release of vendor software is being imported).

Note that you do not check out a vendor module in order to import it -- a **cvs import** can determine directly from the repository what changes have been made and add new versions to the end of the vendor branch as appropriate. This is what distinguishes **cvs import** from **cvs ci**.

7.4.4 What is this vendor tag?

The vendor tag is a “special” branch tag. It always applies to branch 1.1.1 of the software. It is not like a normal CVS branch tag (it is not a “magic” branch tag for example, and does not have the form 1.1.0.1), but rather it is an RCS branch tag.

CVS branch tags differ from RCS branch tags in that CVS branch tags can point to non-existant branches, while RCS branch tags will always point to a branch where at least one revision exists. For example, if there is a CVS branch tag 1.2.0.2 there may or may not be a revision 1.2.2.1 to apply this tag to. If there is an RCS branch tag 1.2.2 then there will always be a revision 1.2.2.1.

The vendor branch tag is the only branch tag that CVS uses that is in the RCS format -- that is allowed partially because whenever a new vendor branch is created, at least one revision will always be placed on the end of this branch.

7.4.4.1 Default Branches

Whenever you examine the **`cvs log`** of a file recently imported by **`cvs import`** you will see the following in the log:

```
.
.
head: 1.1
branch: 1.1.1
.
.
```

The “head” entry indicates that the latest revision on the trunk is revision 1.1. The “branch” entry indicates to CVS (and RCS) that the “default” version to check out is on the 1.1.1 branch, and not the main trunk. This is what causes CVS to fetch revision 1.1.1.1 on a **`cvs co`**, and not 1.1.

7.4.4.2 Resetting the default branch

In many cases it will pay to reset the default branch to the vendor branch after a successful import of a new version of software. This will have the effect of making sure that software developers who check out the new version of the product get the latest version from the vendor branch rather than the main trunk.

The command to do this is:

```
cvs admin -b<vendor-tag> <module-name>
```

7.4.4.3 Generating multiple vendor branches

DANGER WILL ROBINSON!! THIS SECTION IS FOR CVS EXPERTS ONLY!!

CVS used to support more than one vendor branch. It doesn’t support that any more, although multiple vendor branches can exist in the same RCS file.

The **`cvs import`** command supports a **`-b <branch-number>`** option. This must be a three number RCS branch ID -- the default branch number used by **`cvs import`** is 1.1.1. You can use a

different branch ID, for example “2.3.1” or “5.7.0” if you like. In most cases there is no need to do this.

In any case, do not use the branch number to reflect the release number of the software. For example, if you are importing release 5.7 of the gruff software into your repository, do not be tempted to insert a “-b 5.7.0” option into the cvs import command. There is no need to do this -- release numbers are not reflected in the revision numbers but in the tag names.

A better way to do this is to use a command like this:

```
cvs import software/vendor/gruff gruff-corporation release-5_7
```

This will import your vendor software onto the default (1.1.1) vendor branch, and tag it with the release-5_7 tag which makes much more sense.

There is only one use for the -b option in **`cvs import`** -- that is when a preliminary release of the vendor software has been imported and it is known in advance that the preliminary release needs to be discarded as soon as the new release is imported. Using a -b option with a different branch number for the second import can make it simpler to write a script to obsolete (using **`cvs admin -o`**) the older revisions and remove them from the repository.

After using a -b option, it may be necessary to reset the vendor branch using **`cvs admin -b`**.

7.4.4.4 Generating multiple vendor tags

There is no reason to generate multiple vendor tags. You should use different release tags for importing the different releases of the same software, but you should use the same vendor tag each time.

For example, if you import release 5.7 of gruff using this command:

```
cvs import software/vendor/gruff gruff-corporation release-5_7
```

Then you should import release 6.0 of gruff using this command:

```
cvs import software/vendor/gruff gruff-corporation release-6_0
```

... using the same vendor tag for the second import as you used for the first.

Using a different vendor tags for a second import merely places a second branch tag on the same branch. This has the effect of cluttering up the revision log.

You should even use the same vendor tag when importing to a different branch (using the -b option). This has the effect of moving the branch tag away from the old branch, effectively “hiding” it from CVS.

7.4.5 Generating Patch Files

It may be useful in some cases to generate patch files. A patch file is a file containing differences between one release of a module and another release of a module. Essentially, it is the output of diff(1) run on every file in a module.

To generate a patch file for a module, a base point and an end point must be established. Each of these can be any of:

- The head version of the module.

- The head of a tagged branch of the module.
- A tagged release of a module.

A patch file between two tagged releases or branches of a module is generated using the following command:

```
cvs rdiff -r <tag-1> -r <tag-2> <module>
```

If the second tag is omitted, then the patch file is generated between the tagged version of the module and the head version of the module. For example:

```
cvs rdiff -r <tag> <module>
```

This can be useful in determining the changes made between a vendor branch and the main branch, using:

```
cvs rdiff -r VENDOR <module>
```

7.5 Software Releases

The release of software to a customer is the end product of any software configuration management activity, and indeed the end point of any software development activity. CVS contains several tools to make the release of software easier and more automated.

7.5.1 Release strategies

There are two different schools of thought that I have encountered regarding the use of tags in CVS. Each of these methods provides a different release methodology.

- Local builds are created from the head revision of each file in a set of modules.
  - Developers add new revisions of files to the head revision of each file. Configuration control exists over the head revisions.
  - Local (in-test) changes are either maintained by the developers in their sandbox directories, or placed on to development branches where they are required by other developers.
  - When a release is to be created, it is created by tagging the head revisions.
- Local builds are created from tagged revisions of each file in a set of modules.
  - Developers add new revisions of files to the head revision of each file. No configuration control exists over the head versions.
  - Developers move tags around within modules to indicate which revision of a file should be included in a build.
  - Local (in-test) changes are either maintained by the developers in their own sandbox directories, or committted to the trunk.
  - When a release is to be created, it is created by re-tagging existing tagged revisions.

7.5.2 Tagging using “rtag”

Immediately prior to a software release, a tagged version of the software to be released should be generated. This can be done on a module by module basis by using the rtag command. The syntax for rtag is as follows:

```
cvs rtag [ -r <revision-tag> ] <release-tag> <module>
```

Without the optional "-r" argument, **cvs rtag** tags the head revision of each file in the module with <release-tag>.

With the -r option, the revisions tagged with <revision-tag> are tagged.

7.5.3 Exporting

As an alternative to **cvs co** when creating a release, the **cvs export** command may be used to "export" the modules from the repository. The syntax for export is as follows:

```
cvs export -r <release-tag> <module>
```

Note that the **-r <release-tag>** option in this case is not optional! A tag name must be supplied for the export command. The reason for this is to ensure that a tag really has been applied to the software to be released so that it can be re-generated in future. This is what is called "creating a baseline".

The **cvs export** command has one other side effect. This is that the [redacted] and other RCS keywords are expanded and then stripped from each file -- this is so that the file can be re-imported into CVS at another site without losing the Id information inserted into the file at this site.

7.5.4 Branching

Although the concept of branching has been discussed earlier, and CVS branches can be applied by any developer, it is worth while reviewing the concept of branching at this point.

At the point of software release (where a tag has been applied to indicate that the software has been released at that point), is a worthwhile place to branch a file. This is so that bug fixes can be applied to that release of the file without affecting the main line of development.

It is often necessary to branch every file in a module at the point of release. This is so that branched versions can then be inserted by the developers in the appropriate place without them having to create the branch tags (this reduces the possibility of error on the part of the developers, whose skills after all are mostly concentrated on the production of code, not on the operation of CVS or SCM policies).

Branching of a module can be done using the cvs rtag command. A branch is created by using the -b option like this:

```
cvs rtag -r <release-tag> -b <branch-tag> module
```

- The release tag in this case will be the tag that was applied to the software when it was tagged for release (using the original **cvs rtag** command).
- The branch tag will be a new magic branch tag that is applied to the module. The **-b** option indicates that this will be a branch tag.

7.5.5 Magic branch tags

Another point worth examining is that a branch tag is not a normal CVS or RCS tag. It is a "magic" tag used to construct a branched RCS branched revision when necessary. It does not point to a real revision of the file, rather it is a "rump" revision that will only be created when a **cvs ci** is done.

7.5.6 Release Summary

In summary, the overall process for releasing software is as follows:

- **`cvs rtag -r release-tag`**
- **`cvs export -r release-tag`**
- make and build
- **`cvs rtag -r release-tag -b branch-tag`** if necessary.

7.6 Summary

In this section I have examined some of the parts of CVS that make SCM happen. This is not intended as a thorough reference to the complete commands of CVS -- the audience should read the CVS manual for more information.

Software configuration management staff that deal with CVS on a day to day basis should become familiar with all of the commands of CVS. The best way to do this is with experimentation, and reading the CVS manual and users' guide.

You should also be familiar with the concepts of configuration management, and it is intended to present these in the next session. CVS is a software configuration management tool, in the same way that an axe is a wood cutting tool -- CVS does not do SCM for you any more than an axe will cut wood for you.

8. Software Configuration Management

This section covers some of the methods and processes involved in Software Configuration Management. Although CVS is strictly not a Configuration Management tool (it is a Version Control tool), it does have some procedures and files that can be used to implement a SCM policy.

CVS does not provide bug tracking, neither does it provide an SCM policy. An SCM policy is best implemented by staff, not by software.

8.1 SCM Policy

An SCM policy is what defines how your organisation will manage software change.

The basis of an effective SCM policy is a revision control system. An SCM policy can be as simple as a procedure in which CVS is used as a revision control system and an understanding by CM staff (documented or otherwise) in how releases are produced and distributed.

An SCM policy should also include the implementation of some kind of problem tracking system. There are plenty of problem tracking systems available, ranging from lots of yellow sticky notes and shouting, to software packages such as GNATS or xpts. Many organisations (far too many, in my opinion) have their own internally developed problem tracking software packages.

A more complex SCM policy (perhaps based on MIL-STD-973) will include elements of Configuration Identification, Configuration Status Accounting, and auditing procedures as well as all of the above items.

8.2 Configuration Identification

> Configuration identification includes the selection of CIs; the determination of the types of configuration documentation required for each CI; the issuance of numbers and other identifiers affixed to the CIs and to the technical documentation that defines the CI's configuration, including internal and external interfaces; the release of CIs and their associated configuration documentation; and the establishment of configuration baselines for CIs.
>
> [MIL-STD-973]

Configuration Identification is the process whereby your organisation selects and determines how the individual components of a software project are to be identified.

You can assist in the basic process of configuration identification by following these procedures:

- Make sure that there exists a modules file in your CVS repository (in the CVSROOT directory). This is what CVS uses to identify each software module.
- Ensure that there is a document somewhere (inside or outside of the repository) that describes each of the software modules, and what its purpose is.
- Put comments into the modules file that refer back to the document, or put extracts from the document into the CVS modules file as comments (remember that "modules" is a text file, not a word processing document, and each comment should be preceded with a hash sign -- "#").

- Ensure that there is an effective release procedure. The release procedure should involve a system of tagging, identification, numbering, and backup. Release procedures are described in more detail later.

8.3 Configuration Control

Configuration Control is:

> The systematic proposal, justification, evaulation, coordiation, approval or disapproval of proposed changes, and the implementation of all approved changes, in the configuration of a CI after establishment of the configuration baseline for the CI.
>
> [MIL-STD-973]

8.3.1 Change Authorisation

Change authorisation is a method, or set of methods, that determines what changes can be made to a piece of software, and who may make that change. Change authorisation is a management procedure.

An example of change authorisation is to vest in the project manager or team leader the power to authorise any changes to a piece of software. The proof of that authorisation can be in the form of a "statement of work" (SOW) that the team must perform. The top level statement of work can be in the form of a contract (or product specification) agreed to between a software developer and a customer.

In an organisation involved in a top-down design methododlogy, that statement of work will then be decomposed into a series of lower level SOWs that are assigned to a number of software teams or software engineers. If changes occur in the top level SOW (for example, the customer desires a new feature to be added to the software), then the lower level SOWs are changed to reflect the higher level change, and authorisations are issued to software engineers to implement that change (reporting on the costs of that change for accounting or billing purposes if necessary).

Alternatively, change authorisation may be vested in a Change Control Board (CCB). The function of the CCB is to authorise changes to lower level items of software, either by inspecting the changes or implementing a policy where changes are authorised by some instrument such as a Work Authority.

CVS does not implement any kind of change authorisation procedure. The establishment and implementation of a change authorisation procedure is the responsibility of the management of an organisation.

8.3.2 Work Authority

A work authority is an instrument that authorises a software change. Typically there are two different types of work authorities, Trouble Reports and Engineering Changes.

A software contract (or specification) can also be considered a Work Authority. This is effectively the base form of a work authority.

8.3.2.1 Software Contracts

Not all software is delivered according to a contract (for example, off-the-shelf packages are produced as a part of a research and development programme rather than as a contract), but a lot of the software requiring Configuration Management is subject to a contract.

In its most basic form, a software contract is an agreement between a software developer (an individual or organisation) and a client. The agreement usually states that the software developer will develop a certain piece of software for the client, and in exchange the client will pay the software developer a certain sum of money. There may be other considerations, such as hardware, services, support, and testing, but I will ignore those for the sake of simplicity here.

A software contract, from the point of view of the developer, is a basic statement of work, and as such can be considered a work authority.

A software contract may have an attached software specification, such as a MIL-STD-498 CSS, that states in clear and precise terms what form and functionality the software will have. This specification can then be further decomposed by the software developer (in the case of MIL-STD-498, into one or more CSRS and CSDDs) into lower level statements of work.

A software contract, in the simplest of terms, may involve a purely verbal agreement. In this case it is often difficult to distinguish between a Trouble Report and an Engineering Change (and for this reason I do not recommend that anyone enters into a verbal software contract of any description).

8.3.2.2 Trouble Reports

Trouble Reports (Software Trouble Reports, Software Problem Reports, Problem Reports, or whatever other name your organisation likes to give them) are basically statements that the software as supplied to a customer does not conform to the specification for that software, or is otherwise defective.

Trouble Reports authorise software changes on the basis that unless the software is changed, the defect will continue to exist. Assuming that you work for an organisation that does not pride itself on delivering defective software, the existence of a Trouble Report can be made into an authorisation for a software engineer to make a specific change to rectify the defect.

In normal circumstances most organisations will not charge a client for the delivery of software changes that involve the repair of Trouble Reports.

8.3.2.3 Engineering Changes

Engineering Changes (Engineering Change Notices, Engineering Change Proposals, Contract Amendments) are a form of work authorisiation that does not imply that the software is defective. Rather, it is an extension to the initial agreement (assuming that there existed one) between the software developer and the client.

The reason Engineering Changes exist is that users of software are either not entirely sure of their requirements in advance, or that the requirements change, or that a client may be so impressed by the software that they see delivered that they want even more.

Being an extension of one part of a software contract (the specification), an Engineering Change is often an extension of another part (the price).

8.3.3 Change Control

Change Authorisation is the process whereby Change Authorities, or Statements of Work, are created and assigned to software engineers or software teams.

Change Control is the process whereby a correlation is made between software changes and the output of the Change Authorisation process. Change control ensures that all software changes made are done in accordance with, or in relation to, a statement of work.

CVS does not by itself implement a change control procedure. It does, however, provide some "hooks" or entry points by which SCM staff may themselves implement change control.

8.4 Implementation of Configuration Control

Having established a change authorisation procedure, your organisation will be in a position to use CVS as a change control mechanism. As stated previously, CVS cannot be used as a change authorisation method, and so at best it provides one half (the machine half) of the configuration control policy that your organisation requires.

The process of implementing configuration control includes the following:

- Restricting access to the CVS repository to allow authorised changes only.
- Ensuring that established product baselines are changed by virtue of work authorities only. A work authority in this case will be a change request derived from a trouble report or an engineering change.

In the CVS repository, there is a directory called "CVSROOT". This directory contains a number of control files used by CVS. These files are site-dependant (that is, they can be modified and maintained by the CVS administrators at each CVS site), and can be used by your site to implement configuration control.

In section 7.5.1 (Release strategies) I examined two possible architectures of release methods using CVS tags. These depend on whether local builds are created from tagged or head revisions from the CVS repository.

If you are building local builds from head revisions, your organisation will need to perform configuration control at the commit stage. If you are building local builds from tagged revisions, your organisation will need to perform change control at the tag stage.

8.4.1 Use of the files in CVSROOT

The files in the CVSROOT directory form a single CVS module. These files may be modified by checking out the module, making changes to the checked out files, and then checking the module back in.

```
cvs co CVSROOT
cd CVSROOT
... (hack away) ...
cvs ci
```

8.4.1.1 modules

The CVS modules file is perhaps the most important of the files in CVSROOT. The file implements a form of Configuration Identification.

Each module in the CVS repository is listed in the modules file. The format of entries in the modules file is as follows:

```
module_name  [options]  module_directory  [file ...]
```

Note that both the [options] and the list of files are optional. A module name may be a pointer to a directory, and then the module is assumed to contain all of the files in this directory. Note that a module can contain a list of file names, however in this case the following points should be noted:

- The module will contain only those files listed on the line in the modules file.
- New files added to the checked out module using “cvs add” will not be added in the modules file. These must be added manually.
- A single directory may contain multiple, possibly overlapping, modules. This is not generally a good configuration management practise, however it can be useful in the case of a subset of a module being of importance to a particular set of developers or documentors.

The options on the command line of a module file can perform various types of checks when a module is accessed. These are as follows:

-d <dname>	Normally, when a module is checked out, it will be placed into a directory called module_name. This option sets the name of the directory into which the module will be checked out to <dname>.
-i <prog>	<prog> will be run whenever the module is checked in to the repository. <prog> is the name of a program, which will be passed the full pathname of the affected directory in the repository.
-o <prog>	<prog> will be run whenever the module is checked out of the repository.
-s <status>	This option assigns a “status” to the module. When cvs co -s is run, the modules will be listed in status order and then in module name order. This option has no other meaning.
-t <prog>	<prog> will be run whenever the module is tagged using “cvs rtag”. There is no way to specify within the module file a program to be run whenever “cvs tag” is run -- see the section on “taginfo” for a method to do that.
-u <prog>	<prog> will be run whenever a “cvs update” command is run from the top level of the checked out module.

Note that many of these options are duplicates of functions that can be performed in other ways. For example, the “commitinfo”, “loginfo”, “editinfo” and “taginfo” files can be used in addition to, or in place of, various of the above options.

8.4.1.2 commitinfo

The **commitinfo** file is used to allow and deny **cvs ci** (check in or commit) operations on the repository. Each line in the commitinfo file is of the following form:

```
<directory pattern>        <program name>
```

Each time a commit is attempted on the repository, a check is made between the directory name of the file being committed, and the directory patterns listed in the **commitinfo** file. If there is a match, then the program listed in <program name> is run.

It is often the case that there is only one check that must be run before committing any file in the repository. The syntax to introduce this is as follows:

```
DEFAULT                    <filter name>
```

Note that any filter listed on the DEFAULT line can be over-ridden by a per-directory line which specifies a different filter.

Alternatively, there may be an extra check that needs to be run when committing any file in the repository, in addition to other file by file checks. The syntax for this is as follows:

```
ALL                        <filter name>
```

The most common way to write these programs is as perl scripts. When a **commitinfo** program is run it is passed the following arguments on the command line:

- The location of the directory (relative to the repository).
- A list of files that are being committed.

These arguments are in some cases unimportant, but may be extracted using standard perl commands. Alternatively a C program can be used, in which case they are provided as argv to the main() function.

Note that the **commitinfo** file is not a program and will not be executed -- it simply names a list of programs that are executed.

If an **commitinfo** program exits with a non zero exit code, then the commit is aborted.

8.4.1.3 rcsinfo

rcsinfo is used to manage revision log information. The lines in the **rcsinfo** file list “templates” that are automatically loaded into the editor when a revision log comment is made.

These templates can prompt the developer for extra information -- for example a bug ID or trouble report number, etc. They can be used as simple notification that a bug ID needs to be entered into the revision log, or can serve as full templates (to be parsed later by editinfo).

rcsinfo supports the same DEFAULT and ALL keywords as described for **commitinfo**.

Note that the entities listed in the rcsinfo file are not scripts -- they are just templates. These should be flat ASCII format text files.

8.4.1.4 editinfo

editinfo is used to define a program to edit the log info messages. This program can contain a post edit check to ensure that the contents of the revision log information entered by the developer are correct. It works in the same way as a **commitinfo** file, except that it names editors to be used for the revision log information. If there is no editor defined for a module in the **editinfo** file, then the environment variables **$CVSEDITOR** and **$EDITOR** are checked for an editor, or **vi** is used.

editinfo supports the same DEFAULT keyword as described for **commitinfo**, however the ALL keyword is not supported.

The programs defined in **editinfo** are passed the full path to the current log message template file as a parameter. A normal **editinfo** script will usually call some standard editor (or check the environment) to edit this file, then perform some post editing checks to make sure that the file has the correct format.

editinfo programs are most commonly used in conjunction with a matching template in the **rcsinfo** file. The CVS users' guide contains an example of an **editinfo** script and a matching **rcsinfo** file.

If an **editinfo** program exits with a non zero exit code, then the commit is aborted.

8.4.1.5 loginfo

loginfo is another file used to manage revision log infomation. **loginfo** contains a list of filters that are run over the revision log information entered by the developer, after the execution of **editinfo**.

loginfo supports the same DEFAULT and ALL keywords as described for **commitinfo**.

The filter is sent revision log information to its standard input. The filter can then deal with the information sent to it in any way it likes, including logging this information to a **commitlog** file (which is what the default log.pl script provided with CVS does).

8.4.1.6 taginfo

The **taginfo** file (only available in CVS versions later than 1.5) works in the same way as the **commitinfo** file, except that it names scripts that are to be run in a tag or rtag operation.

taginfo supports the same DEFAULT and ALL keywords as described for **commitinfo**.

The following arguments are passed to a script named in the **taginfo** file:

- The tag name.
- The location of the directory (relative to the repository).
- A list of files that are being committed.

If a **taginfo** program exits with a non zero exit code, then the tag or rtag is aborted.

8.4.1.7 cvsignore

A **cvsignore** file in the **CVSROOT** directory works like a global **.cvsignore** file -- it lists a number of file patterns that are never to be added to the repository.

This is useful to list certain file types and patterns that should not be committed. If a file matching one of the patterns listed in a **cvsignore** file is found by **cvs ci**, **cvs add** or **cvs update**, it is ignored.

8.4.2 Control of commits

To implement change control at the commit stage, the following procedure is recommended:

- Create an **rcsinfo** file which points to one or a number of revision log templates. Expect that a developer, when commiting a change to the repository, will follow instructions as listed on the template.
- Create an **editinfo** file to verify that the information entered against the log template is correct. For example, the programs listed in the editinfo file could examine a TR database to determine whether the trouble report listed in the revision log is in fact a legitimate (and open) TR according to the database. Have each of these programs return a non-zero exit status (preferably -1) when an error occurs.
- If necessary, create a **commitinfo** file to verify the cvs commit procedure. Within this commitinfo file, define a program (or a number of programs) that determine whether the commit should take place or not. These programs can actually read through the file that is being committed, and verify that it conforms to coding standards, contains history information in the correct format, contains comments stating the TR number of the change, etc.

The complexity of the programs and templates used for commit control can vary widely.

8.4.3 Control of tagging

To implement change control at the tag stage, the following procedure is recommended:

- Create a **taginfo** file which verifies the tag operation. This file should list programs that accept a trouble report or change number (perhaps as input) and verify this against a database. This should work in much the same way as the editinfo program described above.
- Ensure that you have adequate logging of tag activities. This should also be done within programs defined by the **taginfo** file (perhaps using the ALL keyword).

Implementing change control at the tagging stage inserts an extra operation into the loop of producing software, and is not my favourite method. However it does have advantages in organisations where there are a large number of software engineers working on a single product, as it allows verification of commits outside of the build procedures.

A better way to implement this in a smaller organisation is by using local branches. This may become unweildy where there are a large number of developers because there will be too many minor branches.

8.5 Release Procedure

The procedure for releasing software follows directly on from the process of change management and configuration identification. CVS does not provide any tools to directly implement a release procedure, however there are some parts of CVS that are directed at the release process.

8.5.1 Release Authorisation

Authorisation of software releases follows directly on from the change authorisation process. Release authorisation is a management process, and not a software process, and so therefore CVS does not implement any kind of release authorisation procedure.

Your organisation should develop a release authorisation procedure that is directly tied to the change authorisation procedure. This should take into account the following components:

- Authorisation of initial releases of software should be bound to a software testing process. Because there is no direct change authorisation procedure for an initial software release (i.e. software changes in an initial release are not all due to bug fixes or specification changes), there is no way to tie the release of an initial release of software to the change authorisation process.
- Authorisation of subsequent releases of software are based on the change authorisation process. There should be some verification taking place that:
    - All changes to the software since the previous release are due to authorised work authorities.
    - All changes to the software since the previous release have been tested, and integration testing has been performed to ensure that fixing a bug in one part of the software does not break a different but related part.
- The Quality Control organisations (both yours and your customers', if they exist) will require involvement in the software release authorisation process.

8.5.2 Release Control

Release control is the process whereby a correlation is made between the release changes and the output of the release authorisation process. It bears the same relationship to release authorisation that change control bears to change authorisation.

A good release control procedure should reference an appropriate change control procedure. In a perfect world, where change control performs 100% of the CM process, release control is virtually unnecessary (also, in a perfect world, software never has any bugs, meets 100% of the requirements and never needs fixing or extending). Unfortunately (or fortunately, depending on who pays your salary), a perfect world never exists. In an imperfect world, the primary aim of release control is to ensure that the change control process has functioned correctly.

CVS does not by itself implement release control. It does however provide some facilities whereby release control can be implemented. Primarily, these are the ability to use the same hooks as change control does.

To implement a release control procedure in CVS, you should take into account the following points:

- Build logging in to your change control procedures. This means that your setup of the modules, commitinfo, taginfo, editinfo, and loginfo files should not only verify that the attempted software revision change is acceptable, but it should also log the change.

- Use the output of the logging process as an input to your release control procedures. Form a file (document, or log) that describes all of the software changes between two software releases. If your logging information produced by change control is in a suitable format, it should be possible to generate this document automatically by referring to the log information (hint: use a perl or awk script to post process the logs).
- Provide the output of the release control programs in a human readable as well as a machine readable format. If the output is large, produce a summary. This should also be achievable using perl scripts. You are here to assist your management, software engineers, and QA staff to produce a product -- not to make life more complex for them.

8.5.3 Version Description Documents

8.6 Configuration Status Accounting

> The purpose of [configuration status accounting] is to assure accurate identification of each CI and delivered unit so that the necessary logistics support elements can be correctly programmed and made available in time to support the CI. An adequate and accurate CSA will enhance program and function manager's capabilities to identify, produce, inspect, deliver, operate, maintain, repair, refurpish, etc., CIs in a timely, efficient, and economical manner in satisfying their assigned responsibilities.
>
> [MIL-STD-973]

I guess that it's quotes like the above that give configuration management a bad name.

Configuration Status Accounting is the process whereby records are maintained, usually in a database, of the current state and historical state of the software product.

CVS does not maintain any databases of any kind, apart from the revision control files in the repository. It therefore does not implement any kind of CSA.

If it is important that CSA information be maintained by your organisation, MIL-STD-973 (especially appendix H) is a good place for further reading.

8.7 Configuration Audits

Configuration audits are activities that attempt to verify the software product against its design or specification requirements.

8.7.1 Types of Audits

There are two types of audits. These are:

- Product audits. These are part of the configuration management process. They include Functional Configuration Audit (FCA) and Physical Configuration Audit (PCA).
- Procedural audits. Thease are a part of the quality assurance process.

8.7.1.1 FCA

> The objective of the FCA shall be to verify the configuration item's and system's performance against its approved configuration documentation. Test data for the FCA shall be taht collected from the test of the configuration of the item that is to be formally accepted or released for production (prototype or preproduction article). If a prototype or preproduction article is not produced, the test data shall be that collected from test of the first production article.
>
> [MIL-STD-973]

A Functional Configuration Audit (FCA) verifies that the software, as produced, meets the requirements specifications. In a word, the software will pass FCA if it works.

An FCA should examine a piece of software with the following criteria:

- Was the software tested?
- Do the test plans for the software cover all of the requirements in the software specification documents (C-SRS, or whatever)?
- Did all of the tests pass?

8.7.1.2 PCA

> The PCA shall be the formal examination of the as-built configuration of a CI against its design requirements.
>
> [MIL-STD-973]

A Physical Configuration Audit (PCA) is the verification that the software has been built correctly. The PCA examines the software's design documents and verifies these against the design and construction process.

A PCA should examine a piece of software with the following criteria:

- Was the software designed correctly (in accordance with the contract)?
- Was the software design matched to the original requirements?
- Does the software match the software design?
- Is the software design process auditable? Are any changes that may have been made to the specification (via ECPs) traceable to appropriate changes in the design? Are these changes then traceable to the software?

8.7.1.3 Procedural Audits

Procedural Audits are usually conducted by Quality Assurance (QA) staff, rather than by SCM staff. The objective of a procedural audit is to verify that the organisation or individual being audited:

- Complies with some previously established standard. This can be an internal company SCM standard, a standard specified in a software contract, or an external standard such as MIL-STD-973.

- Documents their compliance by specifying procedures that will be followed in order to comply with the standard.
- Follows procedures as laid out in their documentation.

Procedural audits can be internal or external. Internal audits are conducted by an organisation's own QA staff. External audits are conducted either by the customer's QA staff, or by a third party organisation.

There is a difference between an audit and a review. An audit is a quality assurance activity that determines whether there are sufficient procedures in place to ensure that an organisation can produce a product of a certain standard. A review is a quality control activity that determines whether the actual product really does conform to that standard. I will not discuss reviews in this tutorial.

An audit is conducted by an auditor that is experienced with the process of auditing. The auditor does not need to have specific skills in the production of a software product, or in configuration management. The auditor attempts to prove the existance or otherwise of procedures that conform to a standard, not whether the final product as produced according to those procedures is fit for purpose or not.

8.7.2 Conducting an Audit

The conduct of an audit is a quality assurance activity, not a configuration management activity. I will therefore not attempt to be specific and detailed in how an auditor should conduct an audit. There are many books and training courses available that deal with these specifics.

Here are some general principles for conducting an audit of configuration management practises:

- Be objective and audit the processes, not the personnel. Do not make statements such as "I don't like your CM practises", rather state that the "CM practises are incorrect because ...". In general, be careful of the use of the word "you" during an audit.
- Audit the process, the documentation of the process, and the adherance to that documentation, in that order. For example:
    - Ensure that the CM processes are up to date with the requirements of that process. Use reference to standards such as MIL-STD-973 or IS-649 in determining this adherance, not your own personal opinions of what good CM practises are. If there are CM practises written in to the contract then ensure that these are what is being covered.
    - Ensure that there is documentation to support the fact that the processes are being adhered to. CM plans, along with CVS manuals and local guidelines help in this area. The documentation may be more specific about exactly how these practises are implemented.
    - Ensure that the actual practises of the SCM staff adhere to what is stated in the documentation.
- Ensure that corrective action requests raised in audits are followed up.

8.7.3 Surviving an Audit

There are some simple rules for surviving a procedural audit. These are:

- Before the audit, ensure that you know what your procedures are. If your procedures are documented, ensure that the documentation and the procedures match.

- Ensure that your procedures (and their documentation) cover all of the points that the auditor requires. In the case of MIL-STD-973, your procedures must cover the following items:
    - Configuration Identification
    - Configuration Control
    - Configuration Status Accounting
    - Configuration Audits (specifically FCA and PCA).
- Answer all of the questions put to you, and only the questions put to you. Do not volunteer any information that is not specifically asked for that might in any way prejudice the outcome of the audit. Remember that the auditor is the person making the decision as to whether you will pass or fail the audit, and the auditor will be objective in making the decision based on the information presented during the audit.
- Tell the truth. If you are attempting to cover up actions during an audit you will get a bigger corrective action placed on you than if you expose an inadequacy in any other way.

9. Migration to CVS

Since I have by now convinced you what a great system CVS is, you're all probably wanting to know how you can get rid of your old system and replace it with CVS. Here's how:

9.1 Moving from RCS to CVS

Migration from RCS to CVS is a relatively painless process. CVS uses RCS for all of its file level version control, and so RCS files can be directly copied into a CVS repository.

In converting from RCS to CVS, make sure that you follow these steps:

- Create a CVS repository. Within this repository you should create an empty directory structure to hold the RCS files.
- Keep a backup copy of your RCS files in their original location just in case there are any mishaps.
- Copy the files into the repository using the standard unix file copy utilities (cp -rp is useful, as is tar).
- Update the modules file in CVSROOT. This can be updated using **`cvs co modules`** or **`cvs co CVSROOT`**, editing the modules file, and checking it back in again. You should make sure that there is an entry in the modules file for every directory (top level and subdirectory) in the repository.
- Test the change by attempting to **`cvs co`** a few of the new modules you have created, and running **`cvs log`** or **`cvs status`** on some of the files you have checked out.

9.2 Moving from SCCS to CVS

SCCS is a file by file revision control system, while CVS is a repository based system. By and large the greatest task in moving from SCCS to CVS is the change in the development mentality from developer controlled source files to a centralised repository based systems. Having said that, the vast majority of SCCS based development sites that I have encountered have already set up a repository of sorts, because doing that is just good SCM practise.

Some important points to note when converting to CVS from SCCS:

- CVS requires other programs to do some of its work -- in particular RCS and Diff are required. Although a diff utility is provided with most operating systems, I recommend that you upgrade to GNU diff (version 2.7 or later) in the process of upgrading to CVS.
- Make sure that you have the latest version of RCS (version 5.7 is recommended).
- Ensure that the developers and management are informed well in advance of the change. This may be a change of toolset, but it also requires a change of development mentality.

In converting from RCS to CVS, make sure that you follow these steps:

- Create a CVS repository. Within this repository you should create an empty directory structure to hold the RCS files.
- Keep a backup copy of your SCCS files in their original location just in case there are any mishaps.

- There is a CVS supplied utility called **sccs2rcs** that will convert all SCCS files in an **SCCS/** directory to RCS files in an **RCS/** directory. This performs most of the work in doing the actual file level conversion.
- Copy the RCS files created by **sccs2rcs** into the repository using the standard unix file copy utilities (cp -rp is useful, as is tar). During this stage, make sure that you don't lose track of your repository structure.
- Update the **modules** file in CVSROOT. This can be updated using **cvs co modules** or **cvs co CVSROOT**, editing the modules file, and checking it back in again. You should make sure that there is an entry in the modules file for every directory (top level and subdirectory) in the repository.
- Test the change by attempting to **cvs co** a few of the new modules you have created, and running **cvs log** or **cvs status** on some of the files you have checked out.

It may pay to perform some pre-planning for the above steps in this order:

- Create the CVSROOT directory structure in advance. Create and update a **modules** file so that you know what directory structure you are working towards. Print out a copy of this structure and keep it handy -- a diagram or two showing the directory tree structure may be useful here, both to the SCM staff and the developers.
- **sccs2rcs** is a lengthy process. If you have an existing repository of SCCS files, you may be able to create a script to do both the **sccs2rcs** conversion and moving the subsequently created RCS files into the repository. Run **sccs2rcs** a few times on test files so that you know what it does.

9.3 Moving from CVS 1.3 to CVS 1.5

There are very few issues involved in moving from CVS 1.3 to CVS 1.5. Note the following points:

- CVS 1.3 and 1.5 revision file formats are compatible -- because RCS does all of the revision control. Similarly, upgrading from RCS 5.5 or 5.6 to RCS 5.7 (also part of the recommended upgrade process) is relatively painless, because the ,v file format has not changed.
- There are a few extensions to files in the CVSROOT area in CVS 1.5.
- CVS 1.5 will now operate in client / server mode. This will be quicker than NFS for some sites with slow network links.

10. tkCVS

10.1 Introduction

tkCVS is graphical interface to CVS. It supports all releases of CVS later than 1.0. It is written in Tcl/Tk, and so it has an OSF/Motif look-and-feel to it.

tkCVS extends the CVS modules file in a backwards-compatible manner by allowing the repository to contain descriptive names of modules as well as just the module code.

tkCVS provides a directory browser with hooks to a text editor (for working in a checked-out directory), and a module browser in which a hierarchical tree of modules can be viewed. Within the module tree, a module can be viewed, and the files within it may be browsed using the file browser.

Modules may be checked out from the repository using the module browser, either by selecting a tag or by using the default (head) revision. tkCVS also has a hook into the "export" facility of CVS, used for release of software intended for customer sites.

tkCVS also provides various printed reports on the structure and content of the repository.

10.2 Where do I get tkCVS?

tkCVS is available by anonymous ftp from ftp.aud.alcatel.com:/tcl/code/tkcvs.tar.gz

10.3 More information on tkCVS

tkCVS also has a man page, which serves as a complete reference. It is available by installing tkCVS on your system, and then typing "man tkcvs".

tkCVS also has an in-built help system on each of the main windows. This paraphrases the appropriate sections of the tkCVS manual.

11. emacs and CVS

11.1 CVS mode

11.2 emacs VC

12. Remote CVS

Cygnus support introduced a “remote” version of CVS -- this version of CVS includes the ability to operate the repository remotely (i.e. not via an NFS mount).

The remote version of CVS has been the main line of the CVS pre-release tree since CVS version 1.4.92, and has been included in the 1.5 release of CVS. This is, therefore, the principal version of CVS currently available.

To operate remote CVS, firstly each user needs an account on the machine containing the repository. The account should have a “.rhosts” file containing the machine name and user id of the machine from which CVS will be run.

The first difference with remote CVS is that the CVSROOT environment variable allows a remote machine name as a specifier. The format of the CVSROOT variable is as follows:

```
machine_name:/directory
```

This indicates to CVS that it is to operate in “client mode”, and instead of accessing the repository by looking up the CVSROOT directory, it uses the rsh protocol to talk to a CVS operating in “server” mode at the other end of the link.

In “server mode” CVS does most of what it normally does, except that it does not create any new directories or files. It returns these files to the CVS client via the rsh protocol. The client then does the task of saving the files in a checkec out directory.

Note that there is no “CVS daemon”. Since CVS uses the rsh protocol, the “in.rlogind” daemon that runs to allow remote logins on unix machines will start the CVS server program.

13. More Information

13.1 CVS documentation

The following documentation exists for CVS, in addition to what has been provided in this course material:

- The CVS man page -- referred to as cvs(1). There is also a man page dealing with the file formats of cvs -- cvs(5). If you are on a unix system then you should print these out and read them, or browse them with "man".

13.2 CVS books

- Per Cedervist, *Version Management with CVS* (release 0.9); Signum Support AB (Linkoping, Sweden: 1993). This is provided as a texinfo file (cvs.texinfo), and as a PostScript file with the CVS source distribution, in the doc/ directory. A copy of this manual is also provided with this course material.
- Cyclic Software, *CVS Client Server*. This is also provided as a texinfo (cvsclient.texinfo) and PostScript file with the CVS source distribution. It describes the implementation of the client-server protocol used in CVS.
- Brian Berliner, *CVS II: Paralellizing Software Development*. This paper describes the initial impetus behind CVS and its early development. It is interesting reading but now largely out of date. It is also provided in the doc/ directory in the CVS distribution as a troff file (cvs-paper.ms).

13.3 CVS on the Internet

CVS has a page on the World Wide Web:

- http://www.winternet.com/~zoo/cvs/

Cyclic Software, who are the current maintainers and support office for CVS also have a page on the web:

- http://cyclic.com/

There is a mailing list for CVS, currently located at "info-cvs@cyclic.com". To subscribe to this list, send e-mail to the address "info-cvs-request@cyclic.com".

13.4 CVS FAQ

There is a Frequently Asked Questions (FAQ) file (including answers) to CVS which even I found helpful. The file can be found on the internet at either of the following locations:

- ftp://ftp.odi.com/pub/users/dgg/FAQ.gz
- http://www.winternet.com/~zoo/cvs/FAQ.txt

A CVS mini-FAQ is posted weekly to the mailing list "info-cvs@cyclic.com".

13.5 References for CVS

- CVS is being used by the FreeBSD OS development team as their source control system. Some of the Free Software Foundation's use CVS as well.
- CVS was used (and extended) by Prisma to control an operating system kernel project of nearly 4 million lines of source code, contained in 17,000 files over 1,000 directories. See the article "CVS II: Parallelizing Software Development" by Brian Berliner in the CVS source files (under /doc).
- CVS has been used by CSC Australia in the Submarine Project for a few years now, commencing with release 1.2 (plans to upgrade the release are in progress). Several modifications have been made to the base release to support such features as tag logging and locking, audit trails, etc.
- CVS release 1.3 is used by Australian Defence Industries (Electronics Division) as the core of its documentation and source code control system.

14. Acknowledgments

Parts of this training manual were extracted from the following sources:

- David G Grubbs (dgg@world.std.com), The *CVS Frequently Asked Questions (FAQ)* file
- Per Cedervist, *Version Management with CVS (release 0.9)*; Signum Support AB (Linkoping, Sweden: 1993).
- Department of Defense (United States of America), *Military Standard Configuration Management* (MIL-STD-973).