

# CVS Distilled

---

A quick guide to using CVS

Dario Alcocer

---



# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>3</b>
<b>2</b>	<b>Basic Usage</b> .....	<b>5</b>
2.1	Preliminaries .....	5
2.2	Setting up CVS .....	5
2.2.1	Setting CVSROOT .....	5
2.2.2	Initializing CVS repository .....	6
2.2.3	Creating initial CVS work directory .....	6
2.3	Initial check-in .....	7
2.3.1	Adding source files .....	7
2.3.2	Importing source files .....	9
2.4	Making simple changes .....	11
2.5	Using symbolic revisions .....	12
2.6	Managing your work directory .....	14
2.7	Making concurrent changes .....	16
2.8	Managing change conflicts .....	18
<b>3</b>	<b>Advanced Usage</b> .....	<b>23</b>
3.1	Using branches .....	23
3.1.1	Tracking changes to a released version .....	23
3.1.2	Integrating third-party releases .....	27
3.2	Conclusion .....	31



Author: Dario Alcocer <alcocer@helixdigital.com>  
\$Revision: 1.8 \$  
\$Date: 2002/08/27 14:34:21 \$



# 1 Introduction

The following document helps explain how to use CVS in a number of common scenarios. This document isn't meant to replace the standard CVS documentation, but rather it serves to introduce CVS.

For users familiar with other version control systems like RCS, the most difficult concept to come to grips with is that CVS does not support file locking. However, this in fact is not a limitation; file locking does not adequately support concurrent development by multiple developers. This limitation of RCS-style file locking is precisely the reason CVS was developed.

In order to get the full value of this tutorial, it is recommended that you repeat the examples as you follow along, using a locally installed copy of CVS. Access to a Unix-style shell is also assumed. While programming knowledge isn't required to follow the tutorial, it will help to be familiar with programming concepts in order to fully appreciate the tutorial examples.

In addition, you'll also need access to a text editor (e.g. `vi`) to make changes to the samples. Alternatively, you can duplicate the changes by using the patch files included in the tutorial distribution. These patches can be applied to the files using the `patch` utility. If you're not familiar with `patch`, you can still view the patch files to see the changes, and then make the changes manually using a text editor.

The shell commands presented in these examples can be used from any shell unless otherwise noted.



## 2 Basic Usage

This chapter will demonstrate basic usage of CVS, using a small example program to motivate the tutorial<sup>1</sup>. Since learning something as complex as CVS requires a bit of experimentation, the example program serves as a "sandbox" where you can play and try out the various features of CVS, without affecting any of your existing source code in the process.

### 2.1 Preliminaries

Before you can begin the tutorial, you must first prepare the sandbox directory structure used for the tutorial:

```
$ T=$HOME/sb # For csh use: setenv T $HOME/sb
$ mkdir $T
$ cd $T
$ mkdir $T/d
```

Notice that the *T* environment variable has been defined; this variable is not required by CVS. However, this variable will make it easier to refer to the tutorial sub-directories further on.

Download the sample files used for the tutorial and place them in your tutorial distribution directory '*\$T/d*'. Once you've done this, you'll be ready to set up CVS.

### 2.2 Setting up CVS

Once the preliminaries are out of the way, three tasks must be completed before you can start the tutorial:

- You must set the *CVSROOT* environment variable to point to the directory that will be used to store the CVS version files, and
- The directory referenced by *CVSROOT* must be set-up for use with CVS
- Your CVS working directory must be set up.

#### 2.2.1 Setting CVSROOT

You need to set the *CVSROOT* environment variable from a shell command line. If you're using a Bourne-compatible shell (e.g. *bash*, *zsh*, *ksh*), use the following command to set *CVSROOT* for the CVS examples:

---

<sup>1</sup> You'll be adding features of questionable utility to the example program to demonstrate various aspects of CVS. If you're a programmer, resist the temptation to focus on the contrived functionality of the sample and instead focus your attention on CVS.

```
$ CVSROOT=$T/r
$ export CVSROOT
```

If you're using a `csh`-compatible shell (e.g. `csh`, `tcsh`), use the following command line instead:

```
% setenv CVSROOT $T/r
```

The `CVSROOT` variable is usually set in your shell's initialization file. However, to make the tutorial simpler to follow, you can set the value directly from the command line.

## 2.2.2 Initializing CVS repository

Once the `CVSROOT` variable has been set, the next step is to create and initialize the directory specified by `CVSROOT`. The directory specified by `CVSROOT` is where the "version database", or *repository* is kept. The following commands will prepare the CVS repository:

```
$ mkdir $T/r
$ cvs init
```

Once this step is complete, you'll need to create a CVS work directory.

## 2.2.3 Creating initial CVS work directory

Finally, once the CVS repository has been set-up, you need to create a CVS work directory; this directory will be where you will check-out files to which you will be making modifications. Use the following command line to create and initialize your work directory:

```
$ mkdir $T/w
$ cd $T/w
$ cvs checkout -l .
cvs checkout: Updating .
```

The `CVS checkout` command creates a 'CVS' directory in the 'w' directory. The 'CVS' sub-directory is used to store various CVS administrative files related to your work directory and its sub-directories. In fact, it's the presence of these sub-directories (and their contents) that make a user directory a CVS "working" directory.

Once you've created your CVS work directory, you're ready to try the CVS examples.

## 2.3 Initial check-in

For the duration of this guide, you'll be working with a small sample program, a 'hello' program written in C, published by a fictional organization named Acme Corporation. The sample program's purpose is not to be a sophisticated programming example, but rather to embody various traits (e.g. the use of sub-directories) present in more complex software. The sample is purposely simple-minded to help you focus on CVS and its use as a software configuration management tool. The 'hello' sample files should be located in the '\$T/d' directory.

Before you can add the 'hello' source code into your CVS repository, you must first unpack the archive:

```
$ tar xzvf $T/d/hello-1.0.tar.gz
hello-1.0/
hello-1.0/src/
hello-1.0/src/hello.c
hello-1.0/doc/
hello-1.0/doc/README
hello-1.0/Makefile
hello-1.0/Changelog
```

Now that the 'hello' sample has been unpacked, there are two ways to add the files to your CVS repository:

- Add files as they're created using the `add` and `commit` commands, or
- Add an existing source tree all at once using the `import` command.

Since both methods work but have different behavior, it's worth looking at each one.

### 2.3.1 Adding source files

The primary method for adding files to your CVS repository is using the `add` and `commit` commands. This method is preferred if you're adding files to an existing project repository.

First, you'll need to add the source code directory to your repository:

```
$ mv hello-1.0 hello # rename source sub-directory first
$ cvs add hello
Directory /home/alcocer/sb/r/hello added to the repository
```

Note that the source code directory is renamed before adding it to your CVS repository. In general, it's bad practice to include version information in your directory or file names; after all, CVS will be tracking the version information for you. Next, add the files stored in the source code directory to your repository:

```
$ cd hello
```

```
$ cvs add Changelog Makefile
cvs add: scheduling file 'Changelog' for addition
cvs add: scheduling file 'Makefile' for addition
cvs add: use 'cvs commit' to add these files permanently
```

At this point, you could use the `commit` command to save these files in the CVS repository. However, it's easier to use a single commit later to add all the files in one step. You'll need to add the other files stored in the `'doc'` and `'src'` sub-directories now:

```
$ cvs add doc
Directory /home/alcocer/sb/r/hello/doc added to the repository
$ cd doc
$ cvs add README
cvs add: scheduling file 'README' for addition
cvs add: use 'cvs commit' to add this file permanently
$ cd ..
$ cvs add src
Directory /home/alcocer/sb/r/hello/src added to the repository
$ cd src
$ cvs add hello.c
cvs add: scheduling file 'hello.c' for addition
cvs add: use 'cvs commit' to add this file permanently
$ cd ..
```

As you can see, the `'hello'`, `'hello/doc'` and `'hello/src'` sub-directories have to be added to the CVS repository using `add` before the files that they contain can be added.

The `commit` command is required to actually add the files to your repository:

```
$ cvs commit -m "Initial revision."
cvs commit: Examining .
cvs commit: Examining doc
cvs commit: Examining src
RCS file: /home/alcocer/sb/r/hello/Changelog,v
done
Checking in Changelog;
/home/alcocer/sb/r/hello/Changelog,v <-- Changelog
initial revision: 1.1
done
RCS file: /home/alcocer/sb/r/hello/Makefile,v
done
Checking in Makefile;
/home/alcocer/sb/r/hello/Makefile,v <-- Makefile
initial revision: 1.1
done
RCS file: /home/alcocer/sb/r/hello/doc/README,v
done
Checking in doc/README;
/home/alcocer/sb/r/hello/doc/README,v <-- README
```

```

initial revision: 1.1
done
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
done
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
initial revision: 1.1
done

```

Note that `commit`'s `-m` option was used; this option is used to specify a check-in comment on the command line. If the `-m` option is not used, the program specified by the `EDITOR`<sup>2</sup> environment variable is run to allow a check-in comment to be provided.

### 2.3.2 Importing source files

An alternate method to add files to a CVS repository is using the CVS `import` command. The CVS `import` command is meant to add a directory tree of source code. As the name suggests, `import` is meant to add third-party source code to your local CVS repository. However, since `import` also allows you to add an entire directory of source code in one step, you can use it to add your own locally developed source code as well.

*NOTE: Before you can try this method in the tutorial, you'll need to first remove the existing repository you've created with `add` and `commit`. The following commands are not recommended in normal usage. They're only used here to allow adding the 'hello' source code a second time.*

```

$ cd $T
$ rm -fr $T/r/hello
$ rm -fr $T/w/hello

```

*Once the existing source code repository and working sub-directory have been cleaned out, you can add the files again, this time using the `import` command.*

Your first step is to extract the sources:

```

$ cd $T/w
$ tar xzvf $T/d/hello-1.0.tar.gz
hello-1.0/
hello-1.0/src/
hello-1.0/src/hello.c
hello-1.0/doc/
hello-1.0/doc/README
hello-1.0/Makefile
hello-1.0/Changelog

```

Notice that you're unpacking the source code in your working directory; however, CVS doesn't require this. In fact, the 'hello' source code could be unpacked in any directory

---

<sup>2</sup> Normally `EDITOR` points to `vi`. Rather than explain `vi` here, it's easier to specify the check-in comment on the command line.

where there is sufficient disk space to do so. Also, note the directory structure created by unpacking the source archive will correspond to the directory structure created by the `import` command.

Now that the source code archive has been unpacked, it can be imported into your CVS repository:

```
$ cd hello-1.0
$ cvs import -m "Imported sources" hello acme hello-1_0
N hello/Makefile
N hello/Changelog
cvs import: Importing /home/alcocer/sb/r/hello/src
N hello/src/hello.c
cvs import: Importing /home/alcocer/sb/r/hello/doc
N hello/doc/README
```

```
No conflicts created by this import
```

The CVS `import` command requires three arguments in the following order: a repository sub-directory, a vendor tag, and a release tag. The last two arguments are meant specifically for third-party source code; they're not as important if adding locally developed source code<sup>3</sup>. However, the vendor and release tags become important if you start using branching to track local and third-party changes to the source code.

The vendor tag is set to `'acme'` to indicate that the `'hello'` sample is published by Acme Corp. Since you're importing the 1.0 version of `'hello'`, the release tag is set to `hello-1_0`, to indicate which release of `'hello'` is being imported. Since CVS doesn't allow tags to contain periods, the underscore is substituted instead.

If you examine the repository directory `'$T/r/hello'`, you'll see that it mirrors the directory structure present in the original source code directory tree.

The CVS `import` command does not convert the source directory into a CVS working sub-directory, so the source directory is useless, as far as CVS is concerned. Since the source code has now been imported, you can safely delete this directory:

```
$ cd ..
$ rm -fr hello-1.0
```

The source code is now stored in the CVS repository. In order to make changes to the source, you must first check out a copy from CVS:

```
$ cvs checkout hello
U hello/Changelog
U hello/Makefile
```

---

<sup>3</sup> By convention, for local source code, the vendor tag is an identifier for your company or group, and the release tag is `start`. Use the `rtag` command if you decide to correct these tags later.

```

cvs checkout: Updating hello/doc
U hello/doc/README
cvs checkout: Updating hello/src
U hello/src/hello.c
$ cd hello

```

## 2.4 Making simple changes

Now that the ‘hello’ sample source has been put into your CVS repository, it’s time to see how to check-in simple changes. The first change you will make will be to modify `main()` to use an ANSI compliant prototype; refer to ‘`hello-1.0-main-proto.patch`’ in the distribution directory ‘`$T/d`’ for the required changes.

Once you’ve made the require changes, you’re ready to check your change in. (Of course, normally you would compile the program first to verify your changes worked. However, this isn’t a C tutorial, so assume that the changes are correct.) However, before you check-in your changes, it’s nice sometimes to review the changes you made so you can write an accurate check-in comment.

The CVS `diff` command allows you to see difference between your working copy and the revision you started with<sup>4</sup>. If you run the CVS `diff` command now, you should see output similar to this:

```

$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.1.1.1
diff -u -b -r1.1.1.1 hello.c
--- src/hello.c 2002/08/05 17:29:19      1.1.1.1
+++ src/hello.c 2002/08/05 18:33:05
@@ -1,6 +1,7 @@
 #include <stdio.h>

-void main(void)
+int main(int argc, char *argv[])
 {
     printf("Hello, world.\n");
+   return 0;
 }

```

---

<sup>4</sup> You may ask: how does CVS know which revision I started with if CVS doesn’t perform file locking? CVS stores the revision of each file in ‘`CVS/Entries`’; this file is created and updated by `checkout`.

Looking at the output of `diff`, you'll see that CVS looked at the `'doc'` and `'src'` directories; most CVS commands recurse into sub-directories. Also, the `'-u'` (unified diff output) and `'-b'` (ignore white-space differences) options of the CVS `diff` command are used to tailor the output.

You're now ready to check-in your changes to your CVS repository. Use the `commit` command to check your changes in:

```
$ cvs commit -m "Made main() ANSI compliant."
cvs commit: Examining .
cvs commit: Examining doc
cvs commit: Examining src
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
new revision: 1.2; previous revision: 1.1
done
```

Notice that the check-in comment refers to a file-specific change. Normally this would be bad, especially since the commit didn't specify any particular files (most CVS commands recurse into sub-directories by default, if no files are specified on the command line.) However, since the commit only applied to the specific file, it's fine in this instance. However, you should be careful when writing your own check-in comments.

## 2.5 Using symbolic revisions

Now that you've made and checked in your first change, it's time to see how you can record the collection of file revisions that correspond to a particular version of source code. CVS, uses dotted numeric revision labels like `1.23`. However, CVS also supports symbolic revision labels, or *tags*, that are like aliases that refer to specific numeric revisions. Tags are particularly useful to refer to a collection of file revisions, each numeric revision being a distinct label, with the same name.

For instance, the current version of the `'hello'` sample is a modified version of `1.0`. Using conventional version labels, this current version could be referred to as `1.0.1`. This version, in turn, consists of revision `1.1.1.1` of `'Changelog'`, `'Makefile'`, and `'README'`, and revision `1.2` of `'hello.c'`. These collection of revisions can all be assigned a single tag; the tag stored for each file will refer to the corresponding numeric revision:

```
$ cvs tag hello-1_0_1
cvs tag: Tagging .
T Changelog
T Makefile
cvs tag: Tagging doc
T doc/README
cvs tag: Tagging src
T src/hello.c
```

In order to see the actual numeric revisions that correspond to the tag `hello-1_0_1`, you can use the CVS `log` command to see the log information for each project file:

```
$ cvs log -r hello-1_0_1 .
cvs log: Logging .

RCS file: /home/alcocer/sb/r/hello/Changelog,v
Working file: Changelog
head: 1.1
branch: 1.1.1
locks: strict
access list:
symbolic names:
hello-1_0_1: 1.1.1.1
hello-1_0: 1.1.1.1
acme: 1.1.1
keyword substitution: kv
total revisions: 2; selected revisions: 1
description:
-----
revision 1.1.1.1
date: 2002/08/05 17:29:19; author: alcocer; state: Exp; lines: +0 -0
Imported sources
=====

RCS file: /home/alcocer/sb/r/hello/Makefile,v
Working file: Makefile
head: 1.1
branch: 1.1.1
locks: strict
access list:
symbolic names:
hello-1_0_1: 1.1.1.1
hello-1_0: 1.1.1.1
acme: 1.1.1
keyword substitution: kv
total revisions: 2; selected revisions: 1
description:
-----
revision 1.1.1.1
date: 2002/08/05 17:29:19; author: alcocer; state: Exp; lines: +0 -0
Imported sources
=====

cvs log: Logging doc

RCS file: /home/alcocer/sb/r/hello/doc/README,v
Working file: doc/README
head: 1.1
```

```

branch: 1.1.1
locks: strict
access list:
symbolic names:
hello-1_0_1: 1.1.1.1
hello-1_0: 1.1.1.1
acme: 1.1.1
keyword substitution: kv
total revisions: 2; selected revisions: 1
description:
-----
revision 1.1.1.1
date: 2002/08/05 17:29:19; author: alcocer; state: Exp; lines: +0 -0
Imported sources
=====
cvs log: Logging src

RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
Working file: src/hello.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
hello-1_0_1: 1.2
hello-1_0: 1.1.1.1
acme: 1.1.1
keyword substitution: kv
total revisions: 3; selected revisions: 1
description:
-----
revision 1.2
date: 2002/08/05 19:16:07; author: alcocer; state: Exp; lines: +2 -1
Made main() ANSI compliant.
=====

```

Examining the output of the `log` command, you can see there is a symbolic names section for each file. In particular, the symbolic name `hello-1_0_1` for each file refers to revision 1.1.1.1, except for `'hello.c'`, where it refers instead to revision 1.2. Later, you'll see why these tags are useful, but for now, it's enough to note that you can recall this specific version of the `'hello'` sample by providing the tag name via a command-line option to CVS.

## 2.6 Managing your work directory

One confusing aspect of CVS is how sub-directories are handled. CVS is designed to deal directly with either files or with *modules*, the sub-directories of the repository directory. If you need to access any sub-directories present in one of your modules, CVS usually requires

you to specify the repository-relative path of the sub-directory. (There are exceptions to this rule; for example, the `update` command allows you to specify sub-directories while in a module directory, but only if the sub-directory is already present in your working directory.)

Assume for the moment that you've been asked to update the sparse documentation for `'hello'`, but you are short of disk space. You'd like to delete the `'src'` sub-directory from your working directory for a while, at least until you finish the documentation. CVS provides a way to remove a sub-directory from your working directory via the `release` command:

```
$ cvs release -d src
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'src': y
```

The `'-d'` option to `release` tells CVS to delete the directory after it's been released. Before CVS actually releases the sub-directory, it checks to see if you have any uncommitted changes that need to be checked in. In this case, no files have been altered or added (if there were any, they'd be listed by `release`), so can safely answer yes and delete the sub-directory.

Days go by, and you've completed the documentation. Now you'd like to continue working with the source, so you try getting it back:

```
$ cd $T/w/hello
$ cvs checkout src
cvs checkout: cannot find module 'src' - ignored
```

CVS is telling you that it doesn't recognize `'src'` as a valid module, which is true, because `'src'` is in fact a sub-directory of module `'hello'`, the only module you have in your CVS repository. What you need to do is specify to CVS is the repository-relative path of `'src'` instead:

```
$ cvs checkout hello/src
cvs checkout: Updating hello/src
U hello/src/hello.c
```

Unfortunately, CVS didn't create the `'src'` sub-directory, but instead created a `'hello'` sub-directory which contains `'src'`. In order to get CVS to create the sub-directory, you'll need to use the `'-d'` option to `checkout`:

```
$ cvs checkout -d src hello/src
cvs checkout: Updating src
U src/hello.c
```

The `'-d'` option tells CVS to put the contents of `'hello/src'` into the `'src'` sub-directory; if the sub-directory doesn't exist, CVS will create it. CVS can now access the source files correctly because it recognizes `'hello/src'` as a correct path relative to the repository directory. Now that you have the source files you need, you can get rid of the extraneous `'hello'` sub-directory:

```
$ cvs release -d hello
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'hello': y
```

## 2.7 Making concurrent changes

It's time to examine a more complicated change scenario: concurrent changes made to a source code project. This scenario will help demonstrate how CVS supports parallel development by multiple developers.

*NOTE: First, you will need to create a duplicate work directory for the 'hello' sample; this additional work directory will simulate another user making concurrent changes:*

```
$ cd $T/w
$ mkdir hello2
$ cvs checkout -d hello2 hello
cvs checkout: Updating hello2
U hello2/Changelog
U hello2/Makefile
cvs checkout: Updating hello2/doc
U hello2/doc/README
cvs checkout: Updating hello2/src
U hello2/src/hello.c
```

*Obviously, having more than one work directory for a repository can be confusing, and in general should not be used, but for the purposes of this tutorial, it's useful.*

Now, assume that you've been asked to add a `clean` target to the 'hello' Makefile. At the same time, another developer has been asked to add a `test` target, which is used to try out the compiled 'hello' program. Refer to '`$T/d/hello-1.0.1-clean-target.patch`' and '`$T/d/hello-1.0.1-test-target.patch`' for the required changes. In order for this part of the tutorial to go as planned, make sure that the `clean` and `test` targets aren't added to the same Makefile. In other words, '`$T/w/hello/Makefile`' should contain the `clean` target, and '`$T/w/hello2/Makefile`' should contain the `test` target.

Once you've completed the changes to both files, move to the second work directory '`$T/w/hello2`' and check-in the changes:

```
$ cd $T/w/hello2
$ cvs ci -m "Added test target" Makefile
Checking in Makefile;
/home/alcocer/sb/r/hello/Makefile,v <-- Makefile
new revision: 1.2; previous revision: 1.1
done
```

There are two things to note here: first, notice that instead of using `commit`, its alias `ci` is used instead, as in "check-in" (CVS commits can be thought of as check-ins.) Secondly,

notice that you can specify files (in this case, a single file) to commit. This can be used to informally group changes, or to make a single check-in comment make sense for a group of files checked in together. This particular check-in simulates the other developer checking in their changes.

Now that the other user has completed their check-in, it's time to try to check in your changes:

```
$ cd $T/w/hello
$ cvs ci -m "Added clean target" Makefile
cvs commit: Up-to-date check failed for 'Makefile'
cvs [commit aborted]: correct above errors first!
```

While performing your check-in, CVS has noticed that the current revision of 'Makefile' is different than the revision you started with. Therefore, CVS concludes that the repository has new changes not incorporated into your working copy (in this case, the additional `test` target.) Before it will allow you to continue with your check-in, it is forcing you to make your copy up-to-date with respect to the repository.

In order to bring your working copy up-to-date, you'll need to use the CVS `update` command:

```
$ cvs update
cvs update: Updating .
RCS file: /home/alcocer/sb/r/hello/Makefile,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into Makefile
M Makefile
cvs update: Updating doc
cvs update: Updating src
```

As the output of the `update` command indicates, the changes added in revision 1.2 of 'Makefile' will be determined, and then will be merged into your working copy of 'Makefile'. (This is why 'Makefile' is prefixed with a `M`, which signifies that a merge took place.) In fact, if you examine 'Makefile', you should see the `test` target that was added.

Now that you've updated your working copy with respect to your CVS repository, you can now retry committing your changes:

```
$ cvs ci -m "Added test target" Makefile
Checking in Makefile;
/home/alcocer/sb/r/hello/Makefile,v <-- Makefile
new revision: 1.3; previous revision: 1.2
done
```

This example went smoothly because the two changed regions in 'Makefile' did not overlap. Because of this, CVS handled the required merging automatically. If the changed

regions had overlapped, CVS could still perform the required merging, albeit with a little help.

## 2.8 Managing change conflicts

Now that you've seen how CVS handles merging of non-overlapping changes, you can try your hand at overlapping changes. You'll be working with 'src/hello.c' again; in '\$T/w/hello/src/hello.c' you'll be adding an optional name argument to 'hello', while in '\$T/w/hello2/src/hello.c' you'll simulate another developer adding an optional greeting argument. Refer the files '\$T/d/hello-1.0.1-name-arg.patch' and '\$T/d/hello-1.0.1-greeting-arg.patch' for the specific changes.

Once you've made the required changes, move to each work directory and review the changes:

```
$ cd $T/w/hello2
$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.2
diff -u -b -r1.2 hello.c
--- src/hello.c 2002/08/05 19:16:07 1.2
+++ src/hello.c 2002/08/06 13:12:00
@@ -2,6 +2,11 @@

    int main(int argc, char *argv[])

-    printf("Hello, world.\n");
+    char *greeting = "Hello";
+
+    if (argc > 1)
+        greeting = argv[1];
+
+    printf("%s, world.\n", greeting);
+    return 0;

$ cd ../hello
$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
```

```
retrieving revision 1.2
diff -u -b -r1.2 hello.c
--- src/hello.c 2002/08/05 19:16:07 1.2
+++ src/hello.c 2002/08/06 13:15:19
@@ -2,6 +2,11 @@
```

```
int main(int argc, char *argv[])

-   printf("Hello, world.\n");
+   char *name = "world";
+
+   if (argc > 1)
+       name = argv[1];
+
+   printf("Hello, %s.\n", name);
+   return 0;
```

As you can see, both changes overlap; they both change `main()`.

To see how CVS handles this situation, move to the alternate work directory and check in the changes:

```
$ cd ../hello2
$ cvs ci -m "Added optional greeting argument" src/hello.c
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
new revision: 1.3; previous revision: 1.2
done
```

Now, move back to your work directory and check in your changes:

```
$ cd ../hello
$ cvs ci -m "Added optional name argument" src/hello.c
cvs commit: Up-to-date check failed for 'src/hello.c'
cvs [commit aborted]: correct above errors first!
```

The commit doesn't work, since you've made concurrent changes. It's time to update your working copy:

```
$ cvs update
cvs update: Updating .
? hello
cvs update: Updating doc
cvs update: Updating src
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.2
retrieving revision 1.3
```

```

Merging differences between 1.2 and 1.3 into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in src/hello.c
C src/hello.c

```

Notice the last line of update's output: 'src/hello.c' is marked with a C prefix. This prefix is telling you that CVS was unable to perform an automatic merge, but instead found a change *conflict*. CVS uses a particular format inside your files to signal a change conflict:

```

$ cat src/hello.c
#include <stdio.h>

int main(int argc, char *argv[])

<<<<<<< hello.c
    char *name = "world";

    if (argc > 1)
        name = argv[1];

    printf("Hello, %s.\n", name);
=====
    char *greeting = "Hello";

    if (argc > 1)
        greeting = argv[1];

    printf("%s, world.\n", greeting);
>>>>>>> 1.3
    return 0;

```

CVS modified your working copy<sup>5</sup> of 'src/hello.c' to include the conflicting changes. Each area of conflict is surrounded by '<' and '>' characters, and the conflicting changes are separated by '=' characters. The changes above the separator are your uncommitted changes, and the changes below are the changes checked into your CVS repository. These markers are meant to aid you while you perform the necessary changes to complete the manual merge.

In this case, you need to keep both features; some tinkering to each change is necessary. In other cases, you may want to delete one or both of the changes. Since CVS cannot determine the intent of the changes, it can only help by delimiting the conflicting areas, deferring the actual decision to you.

Now, resolve the conflict by making your argument `argv[2]`, and modifying the `printf` call to display `name` (don't forget to remove the conflict markers too.) Once you've made the changes, you should review them:

---

<sup>5</sup> Your original working copy is saved to a separate file; see the CVS manual for details.

```

$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.3
diff -u -b -r1.3 hello.c
--- src/hello.c 2002/08/06 13:26:26 1.3
+++ src/hello.c 2002/08/06 14:12:54
@@ -2,11 +2,16 @@

    int main(int argc, char *argv[])

+   char *name = "world";
+   char *greeting = "Hello";

+   if (argc > 1)
+       greeting = argv[1];

-   printf("%s, world.\n", greeting);
+   if (argc > 2)
+       name = argv[2];
+
+   printf("%s, %s.\n", greeting, name);
+
+   return 0;

```

Now that you've resolved the conflict, you can check in your changes:

```

$ cvs ci -m "Added optional name argument" src/hello.c
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
new revision: 1.4; previous revision: 1.3
done

```

Now that the merge is complete, it's a good idea to tag the version that you have now:

```

$ cvs tag hello-1_0_2
cvs tag: Tagging .
T Changelog
T Makefile
cvs tag: Tagging doc
T doc/README
cvs tag: Tagging src

```

```
T src/hello.c
```

What you've seen so far probably covers a majority of what CVS is used for in daily usage. However, there are other common CVS tasks that deserve to be covered. While these additional tasks don't occur on a daily basis, when they do occur they're often critical tasks that must be completed quickly and efficiently.

## 3 Advanced Usage

Now that you've become familiar with the basic usage of CVS, it's time to move to more advanced uses.

### 3.1 Using branches

One common theme that comes up is that of branching support in CVS. Branching is used to manage the parallel development process in a logical fashion. However, it's important to point out that CVS merely provides a convenient mechanism for branching; it does not mandate policy decisions that detail how branching is used to manage parallel development.

Policy decisions are beyond the scope of a tutorial; it's better to see some of the possibilities to understand how branching can help with common development scenarios. (It's up to you and your team to develop policies that fit your team's working style.) For now, it's time to explore some of these scenarios.

#### 3.1.1 Tracking changes to a released version

One very common use of branching is accommodating modifications to a released version. Branching in this case allows the changes to be stored on a separate line of development, isolated from any experimental code in the current version of the program. CVS allows these changes to be merged later to a future version.

Coming back to the 'hello' sample, assume for the moment that you've been asked by a customer, WidgetCo, to add support for supplying the greeting via an environment variable. The customer is using the 1.0.1 version of 'hello', but the current version is now 1.0.2, which has not been completely tested. Now, you probably don't want the customer using an untested release, but you probably don't want them to wait until the current release is finished either. The answer is to create a new line of development that is based on the source used for version 1.0.1, where you can make additional changes required by this and other customers. This new line of development could also be used to track future bug fixes to 1.0.1.

A branch can be created in CVS using the `rtag` command. The tag name establishes the branch name used by CVS. Now that you've been requested to implement this new feature, you can start by creating a new branch to store your changes:

```
$ cvs rtag -b -r hello-1_0_1 hello-1_0_1-branch hello
cvs rtag: Tagging hello
cvs rtag: Tagging hello/doc
cvs rtag: Tagging hello/src
```

It's a good idea to have the branch tag include the `-branch` suffix, to distinguish it from a normal tag used as a symbolic revision. If possible, the portion before the suffix should indicate the version used as the starting point for the branch.

Now that the branch has been created in the CVS repository, you can check out the code to begin making the required changes (the `co` is an alias for `checkout`):

```
$ cd $T/w
$ cvs co -r hello-1_0_1-branch -d hello-br hello
cvs checkout: Updating hello-br
U hello-br/Changelog
U hello-br/Makefile
cvs checkout: Updating hello-br/doc
U hello-br/doc/README
cvs checkout: Updating hello-br/src
U hello-br/src/hello.c
$ cd hello-br
```

The `-r` option for `checkout` can be used to specify a particular revision. In this case, a branch name was given instead of tag or revision number. If a branch name is used with `-r`, CVS uses the most recent file revisions on the specified branch. The `-d` option tells CVS to check out the code into the specified directory; the directory is created if it doesn't exist.

You can view which branch your working directory is using by using the CVS `status` command:

```
$ cvs status
cvs status: Examining .
=====
File: Changelog          Status: Up-to-date

Working revision: 1.1.1.1 Mon Aug 5 17:29:19 2002
Repository revision: 1.1.1.1 /home/alcocer/sb/r/hello/Changelog,v
Sticky Tag: hello-1_0_1-branch (branch: 1.1.1.1.2)
Sticky Date: (none)
Sticky Options: (none)

=====
File: Makefile          Status: Up-to-date

Working revision: 1.1.1.1 Mon Aug 5 17:29:19 2002
Repository revision: 1.1.1.1 /home/alcocer/sb/r/hello/Makefile,v
Sticky Tag: hello-1_0_1-branch (branch: 1.1.1.1.2)
Sticky Date: (none)
Sticky Options: (none)

cvs status: Examining doc
=====
File: README            Status: Up-to-date
```

```

Working revision: 1.1.1.1 Mon Aug 5 17:29:19 2002
Repository revision: 1.1.1.1 /home/alcocer/sb/r/hello/doc/README,v
Sticky Tag: hello-1_0_1-branch (branch: 1.1.1.1.2)
Sticky Date: (none)
Sticky Options: (none)

```

```
cvs status: Examining src
```

```
=====
File: hello.c          Status: Up-to-date
```

```

Working revision: 1.2 Mon Aug 5 17:29:19 2002
Repository revision: 1.2 /home/alcocer/sb/r/hello/src/hello.c,v
Sticky Tag: hello-1_0_1-branch (branch: 1.2.2)
Sticky Date: (none)
Sticky Options: (none)

```

Looking at the output, you'll notice that there is a "sticky" tag; CVS will save the branch name that corresponds to the revision specified via '-m'. CVS does this by storing the branch name in each 'CVS' directory present in your working sub-directories. Any future updates or commits that are run while inside of the work directory '\$T/w/hello-br' will be relative to the hello-1\_0\_1-branch branch. This keeps the changes made here isolated from the main trunk used for the current version.

'\$T/d/hello-1.0- getenv.patch' to review the changes. Once you've completed the changes, use the CVS diff command to review them:

```

$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.2
diff -u -b -r1.2 hello.c
--- src/hello.c 2002/08/05 19:16:07 1.2
+++ src/hello.c 2002/08/06 23:31:50
@@ -1,7 +1,14 @@
 #include <stdio.h>
+#include <stdlib.h>

int main(int argc, char *argv[])

-   printf("Hello, world.\n");
+   char *msg;
+
+   msg = getenv("HELLOMSG");

```

```

+   if (msg == NULL)
+       msg = "Hello, world.";
+
+   printf("%s\n", msg);
+   return 0;

```

Notice that the differences shown do not include any of the new code added for version 1.0.2; CVS knows that the working directory ‘\$T/w/hello-br’ is not using the main trunk.

Now that the requested feature has been implemented, it’s time to check it in:

```

$ cvs ci -m "Added HELLOMSG environment variable requested by WidgetCo."
cvs commit: Examining .
cvs commit: Examining doc
cvs commit: Examining src
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
new revision: 1.2.2.1; previous revision: 1.2
done

```

If you examine the output of the commit, you’ll see the branching that took place for ‘src/hello.c’. The previous revision was 1.2, while the check-in was assigned revision 1.2.2.1. To see how this new revision was assigned, you’ll need to examine the tags used:

```

$ cvs status -v src/hello.c
=====
File: hello.c           Status: Up-to-date

Working revision: 1.2.2.1 Tue Aug 6 23:31:50 2002
Repository revision: 1.2.2.1 /home/alcocer/sb/r/hello/src/hello.c,v
Sticky Tag: hello-1_0_1-branch (branch: 1.2.2)
Sticky Date: (none)
Sticky Options: (none)

Existing Tags:
hello-1_0_1-branch      (branch: 1.2.2)
hello-1_0_2             (revision: 1.4)
hello-1_0_1            (revision: 1.2)
hello-1_0               (revision: 1.1.1.1)
acme                   (branch: 1.1.1)

```

Remembering that the hello-1\_0\_1-branch branch is based on version hello-1\_0\_1, and recalling the corresponding revision numbers for the branch and the version, you can probably see how the revision number is constructed for the checked in version. The first two numbers 1.2 signify the base revision of branch 1.2.2, which now contains your changes checked in as the first revision branch, hence the actual revision number 1.2.2.1.

Once you've completed the changes for your hypothetical customer, you can celebrate now by tagging the release:

```
$ cvs tag hello-1_0_1-p1
cvs tag: Tagging .
T Changelog
T Makefile
cvs tag: Tagging doc
T doc/README
cvs tag: Tagging src
T src/hello.c
```

Now that you're done with the 'hello-br' working directory, you can get rid of it using the CVS `release` command. The command will prompt you to confirm you want to proceed:

```
$ cd ..
$ cvs release -d hello-br
? hello
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory 'hello-br': y
```

Examining the output before the prompt, you'll see that 'hello' is listed with a question mark; CVS is telling you that the 'hello' program is not checked into CVS. It does this to warn you that if you decide to delete this working directory, you may be losing a file that should be checked into CVS. Since binary programs aren't usually checked into CVS, this warning is fine. Later, you'll see how to tell CVS to ignore object files and program files. For now, you can answer yes and delete the repository.

This branching example shows how you can track source code changes required by customers, while keeping your current source code isolated from these changes. CVS also helps you manage changes made to your software suppliers, again isolating your current sources from potentially disruptive changes, until you're ready to merge the new code into your main trunk.

### 3.1.2 Integrating third-party releases

Another common branching scenario is updating locally modified, vendor-supplied software so that your local modifications are retained. CVS allows you to control how and when vendor-supplied releases are merged into your main trunk, using the sophisticated automatic merging you've already seen.

Assume that you've been notified by Acme that release 1.1 of 'hello' is now available. You'd like to merge their latest release soon because your customers require the advanced features it has. However, you'd like to keep the enhancements you've made too. You can fulfill both requirements with CVS. The new release can be placed on the vendor branch you created when you first checked in the 'hello' source. Once the new release is checked into the `acme` vendor branch, it can be merged later into your main trunk, when you have the time to perform the merge.

```

$ cd $T/w
$ tar xzvf $T/d/hello-1.1.tar.gz
hello-1.1/
hello-1.1/src/
hello-1.1/src/hello.c
hello-1.1/doc/
hello-1.1/doc/README
hello-1.1/Makefile
hello-1.1/Changelog
$ cd hello-1.1
$ cvs import -m "hello-1.1 released by Acme Corp." hello acme hello-1_1
U hello/Makefile
U hello/Changelog
cvs import: Importing /home/alcocer/sb/r/hello/src
C hello/src/hello.c
cvs import: Importing /home/alcocer/sb/r/hello/doc
U hello/doc/README

1 conflicts created by this import.
Use the following command to help the merge:

cvs checkout -jacme:yesterday -jacme hello

```

The conflict message CVS displays is letting you know that when you perform the merge, you will have a change conflict to resolve in `src/hello.c`. This is because the new 1.1 version has changes that overlap with the changes stored in the main trunk. It didn't complain about `Makefile`, because even though you changed this file too, there are no overlapping changes.

As before, once the import takes place, you have no need to keep the source code used to perform the import, and you can safely delete it:

```

$ cd ..
$ rm -fr hello-1.1

```

At this point, the source code for version `hello-1_1` has safely been imported, at the head of the `acme` vendor branch. You can decide to postpone the merging of `hello-1.1` until you have the time to look at merging the newest release with your local enhancements. You aren't required to do the merge immediately after the import is done. So, assume that days go by before you find the time to perform the merge.

Now that you have some time, you'd like to merge the new release into your main trunk. Now, go ahead and perform the merge in a separate directory, `hello-merge`, so that you don't have to deal with your current changes at the same time you're trying to perform the merge. First, check out the current version of `hello` stored in the CVS repository:

```

$ cvs co -d hello-merge hello
cvs checkout: Updating hello-merge

```

```

U hello-merge/Changelog
U hello-merge/Makefile
cvs checkout: Updating hello-merge/doc
U hello-merge/doc/README
cvs checkout: Updating hello-merge/src
U hello-merge/src/hello.c
$ cd hello-merge

```

Next, tell CVS to merge the changes between version 1.0 and version 1.1 of ‘hello’ into the current version of ‘hello’ that you’ve just checked out:

```

$ cvs update -jhello-1_0 -jhello-1_1
cvs update: Updating .
cvs update: Updating doc
cvs update: Updating src
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.1.1.1
retrieving revision 1.1.1.2
Merging differences between 1.1.1.1 and 1.1.1.2 into hello.c
rcsmerge: warning: conflicts during merge

```

Notice the conflict warning; this is the conflict `import` was warning you about before. You can use `update` to see which files have conflicts:

```

$ cvs update
cvs update: Updating .
cvs update: Updating doc
cvs update: Updating src
C src/hello.c

```

The new version of ‘hello’ includes support for a ‘-l’ option to specify the language the greeting is in. You’ll want to merge this feature by having ‘-l’ override any greeting or name specified on the command line. After you make the changes, review them:

```

$ cvs diff -ub
cvs diff: Diffing .
cvs diff: Diffing doc
cvs diff: Diffing src
Index: src/hello.c
=====
RCS file: /home/alcocer/sb/r/hello/src/hello.c,v
retrieving revision 1.4
diff -u -b -r1.4 hello.c
--- src/hello.c 2002/08/08 15:04:37 1.4
+++ src/hello.c 2002/08/08 18:11:11
@@ -1,10 +1,49 @@
 #include <stdio.h>
+#include <string.h>

```

```

#include <unistd.h>

+struct langmap
+{
+    char *code;
+    char *msg;
+};
+
+struct langmap langtbl[] =
+{
+    { "en", "Hello, world" },
+    { "es", "Hola, mundo" },
+    { "it", "Saluti, mondo" },
+    { "fr", "Bonjour, monde" },
+    { "de", "Hallo, Welt" }
+};
+
+#define LANGTBL_SIZE (sizeof(langtbl) / sizeof(langtbl[0]))
+
int main(int argc, char *argv[])
{
+    int i, msgi = 0;
+    char *name = "world";
+    char *greeting = "Hello";

+    if (getopt(argc, argv, "l:") != -1)
+    {
+        for (i = 0; i < LANGTBL_SIZE; i++)
+        {
+            if (strcmp(optarg, langtbl[i].code) == 0)
+            {
+                msgi = i;
+                break;
+            }
+        }
+        if (i == LANGTBL_SIZE)
+        {
+            fprintf(stderr, "hello: error: unsupported language code '%s', "
+                "using default language\n", optarg);
+        }
+        printf("%s.\n", langtbl[msgi].msg);
+    }
+    else
+    {
+        if (argc > 1)
+            greeting = argv[1];

@ -12,6 +51,7 @
        name = argv[2];

```

```

+     printf("%s, %s.\n", greeting, name);
+   }

    return 0;
  }

+   {
+     if (argc > 1)
+       greeting = argv[1];

@ -12,6 +51,7 @
+       name = argv[2];

+     printf("%s, %s.\n", greeting, name);
+   }

    return 0;
  }

```

Now that the merge is complete and working, check your changes in and tag your new release:

```

$ cvs ci -m "Merged hello-1.1 release."
cvs commit: Examining .
cvs commit: Examining doc
cvs commit: Examining src
Checking in src/hello.c;
/home/alcocer/sb/r/hello/src/hello.c,v <-- hello.c
new revision: 1.5; previous revision: 1.4
done
$ cvs tag hello-1_1_1
cvs tag: Tagging .
T Changelog
T Makefile
cvs tag: Tagging doc
T doc/README
cvs tag: Tagging src
T src/hello.c

```

## 3.2 Conclusion

Hopefully this short tutorial has introduced you to enough of CVS that you're able to use it productively on a daily basis. Of course, there is more to CVS than a short tutorial can cover, and you are encouraged to review the manual to become more familiar with it.

