RCS changes:

The following new programs have been added to the rcs set:

branch
name_version
snapshot
rcsstat
branchmerge

The following commands have been modified:

rcsci
rcsco

There were two primary changes.

First, a configuration facility has been added, to allow users to better describe a set of sources with which they want to work. The configuration specification can either be entered on a command line as an extention to the -r option, or can be made permanently in a file called .rcsconfig in the directory where the rcs commands are to be run.

When doing rcs operations on a file, the configuration specification is used to select which version of a file to work with. If the specification is made in a command line option, it takes precedence. If no command line specification is used, the contents of the .rcsconfig file is used. If neither exists, rcs works as normal (usually working with the latest version on the rcs trunk).

The configuration file consists of lines separated by semicolons or newlines. Lines are processed sequentially until a specification is found that matches some version within the rcs element.

Lines may specify branch or version names, dates, the names of files containing element, revsion number pairs, or wildcard specifications.

```
The forms of these are:

branch name     ::=             [a-z0-9.]+
version name    ::=             [a-z0-9.]+
date            ::=             <{rcs date spec}
                                <>
file spec       ::=             [.~/]{unix file name}

wildcard spec   ::=             -for {wildcardspec} {revno}
```

Some examples of revsion configurations might be:

```
        dougs_branch
        ../MACHSUN3/SNAPSHOT.1.21
        <>
---
        -for ../kern/device.c 1.4
        dougs_branch
        <88/01/21 18:40
---
        rcsco "-rdougs_branch;<>" blah.c
---
        make "COFLAGS=-r<88/01/22"
```

To support this sort of thing, there are the snapshot and name_version commands.

By default, snapshot takes a list of directory names as its argument and an optional configuration spec and writes out a file called RCSSNAP. That file contains a list of rcs filename, revision number pairs that correspond to the given configuration spec. It moves any existing RCSSNAP into the file RCSSNAP.old. If given the -u option, it produces a file called RCSSNAP.upd, which contains the rcsco commands that would be required to change the previous version (found in the RCSSNAP.old) snapshotted into the current version (found in RCSSNAP). snapshot can also be directed to put its output into a specific file with the -f option.

An example of how to use the snapshot command would be:

```
snapshot ../{kern,bsd,vm,vax*}
```

The advantage of snapshotting is that you can produce a tangible set of names for a consistent view of sources kept in rcs. That set of names can also be put under rcs control. It is largely independent of the rcs database, so you don’t have to remember dates or worry that some other developer might accidentally remove a symbolic name from the rcs database.

name_version takes a list of file names and an optional configuration spec and assigns a symbolic name to all versions of the file matching that spec.

An example of how to use the name_version command would be:

```
name_version -nXM13 RCS/*
```

The branch command is used to make development easier by providing a convenient way to create and assign symbolic names to branches.

The standard way that development might be done when several parallel development paths are underway should be that each developer works of his own branch or branches from the mainline of development. The developer will probably want to make all brances from some consistent set of sources (this can be facilitated by using the rcs configuation facility described above). When a set of changes is made and working, branched development can be merged into the mainline.

The branch command has the following form:

```
branch [-rrev] [-l] [-u] [-ddate] -nname file [...]
```

The -l and -u options specifiy that the branched file is to be left in the current directory, either locked for modification, or unlocked and read-only. The branch command obeys the standard rcs configuration specifications, so if there is a .rcsconfig file in the current directory and no explicit revision spec is given, it will choose its branch point using the criteria found in .rcsconfig (which is what you want).

Examples of branching are:

```
branch -ndev_io locore.s

branch -r1.3 -ndev_io locore.s

branch -rdev_io -ndougs_dev_io locore.s
```

Once a developer has reached a point where everything he has changed works, he can merge his changes back onto the mainline of development (or some other branch) by using the branchmerge commands.

The branchmerge command has the following form:

```
branchmerge [-v] [-i] -rrev1 [-rrev2] file
```

branchmerge merges two rcs versions of a file, using the changes between those versions and a common ancestor to produce the new version. If only one revision is specified, branchmerge will merge it with the mainline of development. If more than one version is specified, branchmerge will merge the first rev with the second.

When conflicts are encountered branchmerge will, if run in interactive mode, print out the conflict and ask the user if they wish to keep one or both of the conflicts.

In verbose mode, branchmerge prints out a summary of what changes are being applied to the file.*

Examples of how to use branchmerge are:

```
branchmerge -rdev_io mach_ipc.c
branchmerge -rdev_io RCS/*,v          # branchmerge will fail if no branch exists
branchmerge -r -rdev_io mach_ipc.c    # merge mainline changes onto my branch
```

Error codes:

Mach error codes are longword values that have, until recently only had to represent a set of kernel errors. Right now, they are a simple list.

As we expand the number of possible errors, it seems like we might want to subdivide the error space a bit to keep the job of organizing error codes manageable.

Rich and I have been using a scheme in user space that's backwards compatible with the existing kernel error codes. It involves splitting the error code into 4 pieces: the system, the subsystem, the warning bit and the code number. The system is the gross classification of the error. Right now, we have kernel, user and server classes. The subsystem is a subdivision of the system. For example, the loader is a subdivision of the user space code.

I have made kernel errors system 0 and subsystem 0, so existing error codes will continue to mean the same thing. Kernel subsystem 3 is for unix errors, so when something returns an errno, it can easily be converted into a form that we can use. Rich and I have been using a library of error routines that should be merged in with the mach_error stuff in some fashion.

```
/*
 * Mach Operating System
 * Copyright (c) 1987 Carnegie-Mellon University
 * All rights reserved.  The CMU software License Agreement specifies
 * the terms and conditions for use and redistribution.
 */
/*
 * File:        sys/error.h
 * Purpose:
 *      error module definitions
 *
 * HISTORY
 *
 *  10-Feb-88 Douglas Orr (dorr) at Carnegie-Mellon University
 *      Created.
 *
 */

#ifndef _ERROR_H_

/*
 *      error number layout as follows:
 *
 *      hi                                                lo
 *      | system(4) | subsystem(12) | warning(1) | code(15) |
 */

typedef long    error_t;

#define err_none                (error_t)0
#define ERR_SUCCESS             (error_t)0

#define err_system(x)           (((x)&0xf)<<28)
#define err_sub(x)              (((x)&0xfff)<<16)
#define err_warning             (1<<15)
```

```
#define err_is_warning(err)     (((err)&err_warning) != 0)
#define err_get_system(err)     (((err)>>28)&0xf)
#define err_get_sub(err)        (((err)>>16)&0xfff)
#define err_get_code(err)       ((err)&0x7fff)


/*      major error systems     */
#define err_kern                err_system(0x0)
#define err_us                  err_system(0x1)
#define err_server              err_system(0x2)
#define err_local               err_system(0xf)

#define err_max_system          0xf

/*      unix errors get lumped into one subsystem  */
#define unix_err(errno)         (err_kern|err_sub(3)|errno)

extern  void    error_print();
extern  char    * error_string();
extern  char    * error_type();


#endif  /* _ERROR_H_ */
```