NAME

diskutil – disk layout program

SYNOPSIS

diskutil clobber_my_disk [*-op*]
diskutil finit [*-op*]
diskutil init [*-p*]
diskutil fs
diskutil parm

DESCRIPTION

A Mach I386 disk has several "control" structures on it: the BIOS partition structure, the VTOC (Mach partition sructure), and the bad block table. These structures must be initialized for Mach to be able to use the disk. Most structures are never modified again.

The **diskutil** program is used to edit all these structures. We will describe **diskutil clobber_my_disk, diskutil finit, diskutil init, diskutil fs,** and **diskutil parm** on this manual page. These commands provide a "simple" interface for disk initialization. Collectively, the three programs, **fdisk, vtoc,** and **badblocks**, allow you to edit all the fields in the disk control structures. They will be described under separate manual pages. (Note: **fdisk, vtoc,** and **badblocks** are actually links to the **diskutil** program and are equivalent to **diskutil fdisk, diskutil vtoc,** and **diskutil badblocks,** respectively.)

diskutil clobber_my_disk [*-op*]

performs the vanilla disk initialization by executing:

1. **diskutil finit** [*-op*]
2. **badblocks -w**
3. **diskutil fs**

After this step, you are ready to load files onto your disk.

diskutil finit [*-op*]

diskutil init [*-p*]

These commands set up the disk structures so that you are ready to run the **badblocks** command. Both will set up the VTOC to specify that the Mach disk geometry is the same as the BIOS geometry and that there are no entries in the Mach partition table. They will also initialize the badblock table to a default allocation.

The -p flag is used to indicate that you want to override the BIOS disk geometry specifications and will provide them interactively.

The -o flag is followed by a number. This number (in sectors) specifies for the beginning of the MACH partition. The flag is normally used if there are bad blocks in the region of the disk that would be used for replacements of bad blocks.

diskutil finit performs the operations above and in addition, it initializes the BIOS sector 0 partition to one active partition that spans the entire disk and is used for Mach. **This command will totally rewrite sector 0. If you have existing BIOS partitions, or want to leave room to add them you must use** *fdisk* **and then** *diskutil init* **to layout your disk.**

diskutil fs

is used after you run **badblocks** to put Mach partitions on the disk to receive your initial Mach distribution:

"c" partition is the entirety of the BIOS Mach partition.

"a" partition begins at the start of the BIOS Mach partition. It is made approximately 11Meg in size ending on a cylinder boundary. This is large enough for the standard "root" binaries and a few kernels.

"d" partition is contiguous with the "a" partition and approximately 50Meg in size ending on a cylinder boundary. This is large enough for the standard "usr" binaries and the X binaries/environment.

"e" partition is contiguous with the "d" partition and occupies the rest of disk.

diskutil parm

Displays the true disk geometry and the disk geometry that BIOS believes. For disks with a large number of cylinders (> 1024), you will probably want to set the Mach disk geometry to the actual drive geometry. A sample output for an HP Vectra follows:

```
There are 1581 cylinders, 12 heads, and 32 sectors per track.
There are 384 sectors per cylinder.
There are a total 607104 sectors or 296 Meg

BIOS THINKS

There are 791 cylinders, 16 heads, and 48 sectors per track.
There are 768 sectors per cylinder.
There are a total 607488 sectors or 296 Meg
```

NOTES

Of the commands described here, **diskutil parm** is nondestructive and can be run as often as you like. The others, **diskutil clobber_my_disk, diskutil finit, diskutil init,** and **diskutil fs** are run at most once. And only if you can live with the defaults that they select.

SEE ALSO

fdisk (8), vtoc (8), badblocks (8)

BUGS

probably

-

NAME

fdisk – DOS partition maintainance program

SYNOPSIS

fdisk [*-iu*]

PROLOGUE

In order for BIOS to boot the kernel, certain conventions must be adhered to. Sector 0 of the disk must contain boot code, a partition table, and a magic number. BIOS partitions can be used to break the disk up into several pieces. BIOS brings in sector 0 and verifies the magic number. It then searches the 4 BIOS partitions described by sector 0 to determine which of them is *active*. This boot then brings in the secondary boot block from the *active* partition and runs it. Under DOS, you could have one or more partitions with one *active*. The DOS **fdisk** program can be used to divide space on the disk into partitions and set one *active*.

DESCRIPTION

The Mach program **fdisk** serves a similar purpose to the DOS program. When called with no arguments, it prints the sector 0 partition table. An example follows:

```
The data for partition 0 is:
systid 2, start 1, size 38249 (18 Meg), bootid 0
        beg: cyl 0/ sector 2/ head 0; end: cyl 50/ sector 0/ head 0
The data for partition 1 is:
MT
The data for partition 2 is:
MT
The data for partition 3 is:
systid 99, start 38250, size 584460 (285 Meg), bootid 80
        beg: cyl 50/ sector 1/ head 0; end: cyl 252/ sector 129/ head 0
```

The disk is divided into two partitions that happen to fill the disk.

start and *size*
    fields provide the start address and size of a partition in sectors.

bootid 80
    specifies that this is the active partition.

sysid is used to label the partition. Mach reserves the magic number 99 decimal (65 in hex).

cyl, *sector*, and *head*
    fields are used to specify the beginning address plus one and end address plus one for the parititon. Note: these numbers are given using BIOS's understanding of the disk geometry. Further, the two high bits of the *sector* fields are used to provide two more bits for the *cyl* field.

The flags *i* or *u* are used to indicate that the paritition data is to be updated. The **fdisk** program will enter a conversational mode. This mode is designed not to change any data unless you explicitly tell it to. **fdisk** selects defaults for its questions to guarantee the above behaviour.

It displays each partition and ask if you want to edit it. If you say yes, it will step through each field showing the old value and asking for a new one. When you are done with a partition, **fdisk** will display it and ask if it is correct. **fdisk** will then procede to the next entry.

> Getting the *cyl*, *sector*, and *head* fields correct is tricky. So by default, they will be calculated for you; you can specify them if you choose.

After all the partitions are processed, you are given the option to change the *active* partition. Finally, when the all the data for the first sector has been accumulated, you are asked if you really want to rewrite sector 0. Only if you answer yes, will the data be written to disk.

The difference between the *u* flag and *i* flag is that the *u* flag just edits the fields as they appear on the disk. While the *i* flag is used to "initialize" sector 0; it will setup the last BIOS partition to use the whole disk for Mach; and make it active.

NOTES

If you hand craft you disk layout, please make sure that the Mach partition starts on a cylinder boundary. A number of decisions made later assume this.

Editing as existing partition will most likely cause you to lose all the data in that partition.

You should run this program interactively once or twice to see how it works. This is completely safe as long as you answer the last question in the negative. There are subtleties that the program detects that are not fully explained in this manual page.

SEE ALSO

diskutil (8), badblocks (8), vtoc (8)

BUGS

probably

-

NAME

badblocks – bad block table maintainance program

SYNOPSIS

badblocks [-*iuw*]

DESCRIPTION

The Mach disk driver supports replacement of bad sectors and bad tracks that have been found on the Mach partition of the disk. **badblocks** is run in an *initialize* mode to find the problem areas. With today's smart disk controllers that do slip sectors, etc, it is not uncommon to find very few reported errors. The example below illustrates just such a case. In fact the error noted was actually a soft ECC error that the disk driver corrected on the fly. It was entered as a bad block manually.

badblocks with no arguments just prints out the bad block tables.

```
The bad sectors/tracks read from the disk were:
bad sectors:   1 used,  17 reserved, at 38284
bad tracks:    0 used,  29 reserved, at 38301  [1479 sectors]
The first available sector for the root file system is 39780

            Bad sectors detected:
                        Cyl  Head  Sector
                  0:    406    3     23
```

The flags *i*, *u*, and *w* are used to indicate that the data is to be updated. The **badblocks** program will enter a conversational mode. This mode is designed not to change any data unless you explicitly tell it to. **badblocks** selects defaults for its questions to guarantee the above behaviour.

The *i* flag makes the choice of parameters as follows. **badblocks** calculates how many of the initial sectors of the unix partition have been used up with the bootstrap, vtoc, etc. It reserves to the next track boundary as bad block replacement sectors. It then reserves to the next cylinder boundary as bad track replacement. (The Mach file system would start then, but ...) The *u* flag allows you to enter whatever values you want. **THIS WOULD NOT BE WISE.**

Then, **badblocks** allows you to manually specify the bad sectors and bad tracks. You will probably want to use this facility, if bad sectors turn up after the disk has been in service for a while.

Finally, **badblocks** asks you if you want to run *verify*. The *verify* phase scans the entire BIOS Mach paritition looking for bad sectors and bad tracks. It will either just read all the blocks or if the *w* command line switch is used read and write all the blocks in the partition. Clearly, the *w* option is destructive and you ONLY want to run it when you are setting up the disk. The way the *verify* works is that as bad sectors are encountered, they are replaced by a sector from the "reserved" pool. If this pool becomes exhausted, the first replacement track is split up into replacement sectors. The replacement tracks ripple forward in the track table to account for the loss of the first track. A similar strategy is used when the track replacement "reserved" pool fills. We push the start of Mach to the next available cylinder and steal all the tracks up to it. After the scan, we ensure that a few replacement tracks and sectors are available if errors turn up as the disk wears out in service.

> The automatic expansion of the disk bad block area only will happen if there are not any Mach partitions on the disk for the badblock area to overlap.

NOTES

You should run this program interactively once or twice to see how it works. This is completely safe as long as you answer the last question in the negative. There are subtleties that the program detects that are not fully explained in this manual page.

SEE ALSO

diskutil (8), fdisk (8), vtoc (8)

BUGS

probably

NAME

vtoc – Mach partition maintainance program

SYNOPSIS

vtoc [*-iu* | *-m <letter>*]

DESCRIPTION

The vtoc describes the disk for Mach. It specifies the disk geometry that the Mach disk driver will use, the location of the bad block table, and lastly the division of the space on the disk into Mach partitions.

The **vtoc** program, displays the disk layout when called with no arguments. An example follows:

```
The static data for the vtoc is:
        cyls 1581/ heads 12/ sectors 32; alt ptr 15360 alt len 2048
        6 UNIX partition entries.
a: start 141696, end 163967, size  22272 ( 10 Meg), flag 200, tag 2
b: MT
c: start      1, end 607103, size 607103 (296 Meg), flag 200, tag 5
d: start 163968, end 264191, size 100224 ( 48 Meg), flag 201, tag 4
e: start 264192, end 607103, size 342912 (167 Meg), flag 201, tag 4
f: start   1920, end 141695, size 139776 ( 68 Meg), flag 201, tag 4
```

For this particular disk on a HP VECTRA, BIOS uses the geometry *cyls* 791, *sectors* 16, and ead 48. We believe that using the true geometry will improve the performance.

alt ptr and *alt len* specify the location and size of the bad block table on disk. (It is probably a mistake to let you see/alter this.)

The standard Mach partitions are labeled "a", through ""h". We show the *start*, *end*, and *size* of each partition. The *end* is actually calculated and not stored. (The flag and tag data are also probably a mistake and should not be visible.)

Note: in the example, the "root" (paritition "a") is in the middle of the disk. The "f" partition maps the beginning of the disk to the "root".

vtoc *-m <letter>*

A paritition corresponds to an area on the disk. Creating a partition does not also make a Mach file system in the area. You must use the Unix program mkfs to organize the space into a file system. Since mkfs is a hard program to use without some cover program, we provide an option to **vtoc**, *-m <letter>*. This will output to "standard out" most of the parameters needed for mkfs. You need only specify the blocksize (4096 or 8192) and frag size (512 or 1024), to complete the mkfs command.

```
vtoc -m a                          yields
mkfs /dev/rhd0a 22272 32 12

vtoc -m f                          yields
mkfs /dev/rhd0f 139776 32 12
```

The flags *i* or *u* are used to indicate that the paritition data is to be updated. The **vtoc** program will enter a conversational mode. This mode is designed not to change any data unless you explicitly tell it to. **vtoc** selects defaults for its questions to guarantee the above behaviour.

First **vtoc** displays the geometry data and asks if you want to change it. If you say yes, **vtoc** will step through each field showing the old value and asking for a new one. When you are done, **vtoc** will display the changed data and ask if it is ok.

Next **vtoc** will walk you through each partition displaying its data and ask if you want to edit the data. If you say yes, it will step through each field showing the old value and asking for a new one. It will ask for the *size* and then the *end*. It uses the *size* to calculate the *end*. But then goes back and uses the answer for the *end* to recalculate the *size*. So you may specify whichever item is easier for you to calculate with. (If you do not fill in the *flag* and *tag*; it will choose reasonable defaults.) When you are done with an entry, **vtoc** will display it and ask if it is ok. **vtoc** will then procede to the next entry. If all the fields are set to zero, the entry becomes empty (indicated by MT) and disappears.

Note: the number of UNIX parition entries is always updated dynamically by ignoring the MT (emtpy) entries at the end of the partition table.

You are not obliged to set up partitions to fill the whole disk initially. You can always define a new one as needed; although you would typically define parititions to fill the entire disk initially.

After all the partitions are processed, you are asked if you really want to write the changes. Only if you answer yes, will the data be written to disk.

The difference between the *u* flag and the *i* flag

is that *u* just edits the fields as they appear on the disk. While *i* is used to "initialize" the partition table:

"c" partition is the entirety of the BIOS Mach partition.

"a" partition begins at the start of the BIOS Mach partition. It is made approximately 11Meg in size ending on a cylinder boundary. This is enough for the standard "root" binaries and a few kernels.

"d" is contiguous with the "a" partition and approximately 50Meg in size ending on a cylinder boundary. This is large enough for the standard "usr" binaries and the X binaries/environment.

"e" partition is contiguous with the "d" partition and to occupy the rest of disk.

Note that "a", "d", and "e" are set up to be contiguous. So if you change the size or start of one partition, the other partitions slide forward. This technique was used to create the partition table shown in the above example.

NOTES

You should run this program interactively once or twice to see how it works. This is completely safe as long as you answer the last question in the negative. There are subtleties that the program detects that are not fully explained in this manual page.

SEE ALSO

diskutil (8), fdisk (8), badblocks (8), mkfs (8)

BUGS

probably

NAME

sup – software upgrade protocol

SYNOPSIS

sup [*flags*] [*supfile*] [*collection* ...]

DESCRIPTION

Sup is a program used for upgrading collections of files from other machines to your machine. You execute *sup*, the *client* program, which talks over the network using IP/TCP to a *file server* and possibly a *name server* process. The name server process determines appropriate *repository* machines from which to request upgrades. The file server process cooperates with *sup* to determine which files of the collection need to be upgraded on your machine.

In performing an upgrade, the file server constructs a list of files included in the collection. The list is sent to your machine, which determines which files are needed. Those files are then sent from the file server. It will be most useful to run *sup* as a daemon each night so you will continually have the latest version of the files in the needed collections.

The only required argument to *sup* is the name of a supfile. It must either be given explicitly on the command line, or the **-s** flag must be specified. If the **-s** flag is given, the system supfile will be used and a supfile command argument should not be specified. The list of collections is optional and if specified will be the only collections upgraded. The following flags affect all collections specified:

-s As described above.

-t When this flag is given, *sup* will print the time that each collection was last upgraded, rather than performing actual upgrades.

-N *Sup* will trace network messages sent and received that implement the *sup* network protocol.

-P Sup will use a set of non-privileged network ports reserved for debugging purposes.

The remaining flags affect all collections unless an explicit list of collections are given with the flags. Multiple flags may be specified together that affect the same collections. For the sake of convience, any flags that always affect all collections can be specified with flags that affect only some collections. For example, **sup -sde=coll1,coll2** would perform a system upgrade, and the first two collections would allow both file deletions and command executions. Note that this is not the same command as **sup -sde=coll1 coll2,** which would perform a system upgrade of just the coll2 collection and would ignore the flags given for the coll1 collection.

-a All files in the collection will be copied from the repository, regardless of their status on the current machine. Because of this, it is a very expensive operation and should only be done for small collections if data corruption is suspected and been confirmed. In most cases, the **-o** flag should be sufficient.

-b If the **-b** flag if given, or the **backup** supfile option is specified, the contents of regular files on the local system will be saved before they are overwritten with new data. The file collection maintainer can designate specific files to be worthy of backing up whenever they are upgraded. However, such backup will only take place if you specify this flag or the **backup** option to allow backups for a file collection on your machine. The backup mechanism will create a copy of the current version of a file immediately before a new copy is received from the file server; the copy is given the same name as the original file but is put into a directory called **BACKUP** within the directory containing the original file. For example, **/usr/sas/src/foo.c** would have a backup copy called **/usr/sas/src/BACKUP/foo.c.** There is no provision for automatically maintaining multiple old versions of files; you would have to do this yourself.

-B The **-B** flag overrides and disables the **-b** flag and the **backup** supfile option.

-d Files that are no longer in the collection on the repository will be deleted if present on the local machine. This may also be specified in a supfile with the **delete** option.

-D The **-D** flag overrides and disables the **-d** flag and the **delete** supfile option.

-e Sup will execute commands sent from the repository that should be run when a file is upgraded. If the **-e** flag is omitted, Sup will print a message that specifies the command to execute. This may also be specified in a supfile with the **execute** option.

-E The **-E** flag overrides and disables the **-e** flag and the **execute** supfile option.

-f A *list-only* upgrade will be performed. Messages will be printed that indicate what would happen if an actual upgrade were done.

-l Normally, *sup* will not upgrade a collection if the repository is on the same machine. This allows users to run upgrades on all machines without having to make special checks for the repository machine. If the **-l** flag is specified, collections will be upgraded even if the repository is local.

-m Normally, *sup* used standard output for messages. If the **-m** flag if given, *sup* will send mail to the user running *sup*, or a user specified with the **notify** supfile option, that contains messages printed by *sup*.

-o *Sup* will normally only upgrade files that have changed on the repository since the last time an upgrade was performed. The **-o** flag, or the **old** supfile option, will cause *sup* to check all files in the collection for changes instead of just the new ones.

-O The **-O** flag overrides and disables the **-o** flag and the **old** supfile option.

-v Normally, *sup* will only print messages if there are problems. This flag causes *sup* to also print messages during normal progress showing what *sup* is doing.

SETTING UP UPGRADES

Each file collection to be upgraded must have a *base directory* which contains a subdirectory called **sup** that will be used by the *sup* program; it will be created automatically if you do not create it. *Sup* will put subdirectories and files into this directory as needed.

Sup will look for a subdirectory with the same name as the collection within the **sup** subdirectory of the *base directory*. If it exists it may contain any of the following files:

when This file is automatically updated by *sup* when a collection is successfully upgraded and contains the time that the file server, or possibly *supscan*, created the list of files in the upgrade list. *Sup* will send this time to the file server for generating the list of files that have been changed on the repository machine.

refuse This file contains a list of files and directories, one per line, that the client is not interested in that should not be upgraded.

lock This file is used by *sup* to lock a collection while it is being upgraded. *Sup* will get exclusive access to the lock file using *flock*(2), preventing more than one *sup* from upgrading the same collection at the same time.

last This file contains a list of files and directories, one per line, that have been upgraded by *sup* in the past. This information is used when the **delete** option, or the **-d** flag is used to locate files previously upgraded that are no longer in the collection that should be deleted.

Each file collection must also be described in one or more supfiles. When *sup* is executed, it reads the specified supfile to determine what file collections to upgrade. Each collection is described by a single line of text in the supfile; this line must contain the name of the collection, and possibly one or more options separated by spaces. The options are:

base=*directory*

The usual default name of the base directory for a collection is described below (see FILES); if you want to specify another directory name, use this option specifying the desired directory.

prefix=*directory*
Each collection may also have an associated *prefix directory* which is used instead of the base directory for references to files within the collection.

host=*hostname*
hostbase=*directory*
System collections are supported by the system maintainers, and *sup* will automatically find out the name of the host machine and base directory on that machine. However, you can also upgrade *private* collections; you simply specify with these options the *hostname* of the machine containing the files and the *directory* used as a base directory for the file server on that machine. Details of setting up a file collection are given in the section below.

login=*accountid*
password=*password*
crypt=*key*
Files on the file server may be protected, and network transmissions may be encrypted. This prevents unauthorized access to files via *sup*. When files are not accessible to the default account (e.g. the **anon** anonymous account), you can specify an alternative *accountid* and *password* for the file server to use on the repository host. Network transmission of the password will be always be encrypted. You can also have the actual file data encrypted by specifying a *key*; the file collection on the repository must specify the same key or else *sup* will not be able to upgrade files from that collection. In this case, the default account used by the file server on the repository machine will be the owner of the encryption key file (see FILES) rather than the **anon** anonymous account.

notify=*address*
If you use the **-m** option to receive log messages by mail, you can have the mail sent to different user, possibly on another host, than the user running the sup program. Messages will be sent to the specified *address*, which can be any legal netmail address. In particular, a project maintainer can be designated to receive mail for that project's file collection from all users running *sup* to upgrade that collection.

backup As described above under the **-b** flag.

delete As described above under the **-d** flag.

execute As described above under the **-e** flag.

old As described above under the **-o** flag.

PREPARING A FILE COLLECTION REPOSITORY

A set of files residing on a repository must be prepared before *sup* client processes can upgrade those files. The collection must be given a *name* and a *base directory*. If it is a private collection, client users must be told the name of the collection, repository host, and base directory; these will be specified in the supfile via the **host** and **hostbase** options. For a system-maintained file collection, entries must be placed into the host list file and directory list file as described in *supfilesrv*(8).

Within the base directory, a subdirectory must be created called **sup .** Within this directory there must be a subdirectory for each collection using that base directory, whose name is the name of the collection; within each of these directories will be a list file and possibly a prefix file, a host file, an encryption key file, a log file and a scan file. The filenames are listed under FILES below.

prefix Normally, all files in the collection are relative to the base directory. This file contains a single line which is the name of a directory to be used in place of the base directory for file references.

host Normally, all remote host machines are allowed access to a file collection. If you wish to restrict access to specific remote hosts for this collection, put each allowed hostname on a separate line of text in this file. If a host has more than one name, only one of its names needs to be listed. The name **LOCAL** can be used to grant access to all hosts on the local network.

crypt If you wish to use the *sup* data encryption mechanism, create an encryption file containing, on a single line of text, the desired encryption key. Client processes must then specify the same key with the **crypt** option in the supfile or they will be denied access to the files. In addition, actual network transmission of file contents and filenames will be encrypted.

list This file describes the actual list of files to be included in this file collection, in a format described below.

scan This file, created by *supscan*, is the list of filenames that correspond to the instructions in the list file. The scan file is only used for frequently-updated file collections; it makes the file server run much faster. See *supservers*(8) for more information.

lock As previously mentioned, this file is used to indicate that the collection should be locked while upgrades are in progress. All file servers will try to get shared access to the lock file with *flock*(2).

logfile If a log file exists in the collection directory, the file server will append the last time an upgrade was successfully completed, the time the last upgrade started and finished, and the name of the host requesting the upgrade.

It should be noted that *sup* allows several different named collections to use the same base directory. Separate encryption, remote host access, and file lists are then used for each collection.

The list file is a text file with one command on each line. Each command contains a keyword and a number of operands separated by spaces. All filenames in the list file are evaluated on the repository machine relative to the host's base directory, or prefix directory if one is specified, and on your machine with respect to the base, or prefix, directory for the client. The *filenames* below (except *exec-command*) may all include wild-cards and meta-characters as used by *csh*(1) including ∗, ?, [...], and {...}. The commands are:

upgrade *filename ...*
The specified file(s) (or directories) will be included in the list of files to be upgraded. If a directory name is given, it recursively includes all subdirectories and files within that directory.

always *filename ...*
The always command is identical to upgrade, except that omit and omitany commands do not affect filenames specified with the always command.

omit *filename ...*
The specified file(s) (or directories) will be excluded from the list of files to be upgraded. For example, by specifying **upgrade /usr/vision** and **omit /usr/vision/exp,** the generated list of files would include all subdirectories and files of /usr/vision except /usr/vision/exp (and its subdirectories and files).

omitany *pattern ...*
The specified patterns are compared against the files in the upgrade list. If a pattern matches, the file is omitted. The omitany command currently supports all wild-card patterns except {...}. Also, the pattern must match the entire filename, so a leading ∗/, or a trailing /∗, may be necessary in the pattern.

backup *filename ...*
The specified file(s) are marked for backup; if they are upgraded and the client has specified the **backup** option in the corresponding line of the supfile, then backup copies will be created as described above. Directories may not be specified, and no recursive filename construction is performed; you must specify the names of the specific files to be backed up before upgrading.

noaccount *filename ...*
The accounting information of the specified file(s) will not be preserved by *sup*. Accounting information consists of the owner, group, mode and modified time of a file.

symlink *filename ...*
The specified file(s) are to be treated as symbolic links and will be transfered as such and not followed. By default, *sup* will follow symbolic links.

execute *exec-command* (*filename* ...)
The *exec-command* you specified will be executed on the client process whenever any of the files listed in parentheses are upgraded. A special token, **%s,** may be specified in the *exec-command* and will be replaced by the name of the file that was upgraded. For example, if you say **execute ranlib %s (libc.a)**, then whenever libc.a is upgraded, the client machine will execute **ranlib libc.a.** As described above, the client must invoke *sup* with the **-e** flag to allow the automatic execution of command files.

include *listfile ...*
The specified *listfiles* will be read at this point. This is useful when one collection subsumes other collections; the larger collection can simply specify the listfiles for the smaller collections contained within it.

The order in which the command lines appear in the list file does not matter. Blank lines may appear freely in the list file.

FILES

Files on the client machine for *sup*:

/usr/cs/lib/supfiles/name.host
list of names of host machines acting as name servers

/usr/cs/lib/supfiles/coll.list
supfile used for -s flag

<*base-directory*>**/sup/**<*collection*>**/last**
recorded list of files in collection as of last upgrade

<*base-directory*>**/sup/**<*collection*>**/lock**
file used to lock collection

<*base-directory*>**/sup/**<*collection*>**/refuse**
list of files to refuse in collection

<*base-directory*>**/sup/**<*collection*>**/when**
recorded time of last upgrade

/usr/<*collection*>
default base directory for file collection

Files needed on each name-server machine for the name server:

/usr/cs/lib/supservers/coll.host
host name list for system collections

Files needed on each repository machine for the file server:

/usr/cs/lib/supservers/coll.dir
base directory list for system collections

<*base-directory*>**/sup/**<*collection*>**/crypt**
data encryption key for a collection. the owner of this file is the default account used when data encryption is specified

<*base-directory*>**/sup/**<*collection*>**/host**
list of remote hosts allowed to upgrade a collection

-

<*base-directory*>**/sup/**<*collection*>**/list**
list file for a collection

<*base-directory*>**/sup/**<*collection*>**/lock**
lock file for a collection

<*base-directory*>**/sup/**<*collection*>**/logfile**
log file for a collection

<*base-directory*>**/sup/**<*collection*>**/prefix**
file containing the name of the prefix directory for a collection

<*base-directory*>**/sup/**<*collection*>**/scan**
scan file for a collection

/usr/<*collection*>
default base directory for a file collection

SEE ALSO

supservers(8)
The SUP Software Upgrade Protocol, S. A. Shafer, CMU Computer Science Department, 1985.

EXAMPLE

<example>

BUGS

The encryption mechanism should be strengthened, although it's not trivial.

HISTORY

10-May-86 Glenn Marcy (gm0w) at Carnegie-Mellon University
Replaced reference to /usr/cmu with /usr/cs.

29-Mar-86 Glenn Marcy (gm0w) at Carnegie-Mellon University
Updated manual entry to version 5.14 of sup.

14-Jan-86 Glenn Marcy (gm0w) at Carnegie-Mellon University
Updated manual entry to version 5.7 of sup.

04-Apr-85 Steven Shafer (sas) at Carnegie-Mellon University
Created.

-