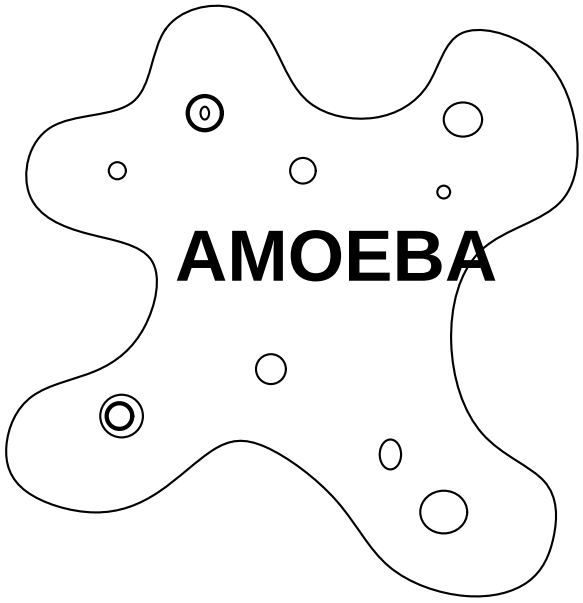
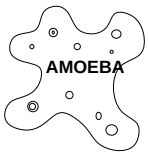
The Amoeba Reference Manual

User Guide



Amoeba is a registered trademark of the Vrije Universiteit in some countries.



is a trademark of the Vrije Universiteit.

Sun-3, Sun-4, NFS, SPARCclassic, SPARCstation, MicroSPARC, SunOS and Solaris® are trademarks of Sun Microsystems, Inc.

SPARC is a registered trademark of SPARC International, Inc.

UNIX is a trademark of Unix International.

Irix and Personal Iris are trademarks of Silicon Graphics Corporation.

DEC, VAX, MicroVAX, Ultrix and DEQNA are registered trademarks of Digital Equipment Corporation.

Intel 386, Pentium, Multibus and Intel are trademarks of Intel Corporation.

Ethernet is a registered trademark of Xerox Corporation.

IBM and PC AT are registered trademarks of International Business Machines Corporation.

X Window System is a trademark of the Massachusetts Institute of Technology.

WD, EtherCard PLUS and EtherCard PLUS16 are trademarks of Standard Microsystems Corporation.

Logitech is a trademark of Logitech, Inc.

Novell is a trademark of Novell, Inc.

3Com and Etherlink II are registered trademarks of 3Com Corporation.

MS-DOS is a trademark of Microsoft Corporation.

Amoeba includes software developed by the University of California, Berkeley and its contributors.

Contents

1 About This Manual

Intended Audience

This manual is intended for use by all people using Amoeba, either by directly logging in or from another operating system running the Amoeba protocol.

Scope

This manual explains the basic concepts of Amoeba from a user's perspective, including how to log in, how to configure the working environment and how to locate required services and objects. It concludes with the collection of reference manual pages for user utilities and servers.

Advice On Use Of The Manual

It is recommended that the introductory material and tutorials be read at least once by all users. For those familiar with UNIX the initial tutorial material may seem straight forward but the file system and directory service are different from that of conventional systems and should be understood before proceeding.

The manual pages are provided for reference and explain how to use the utilities and servers provided with Amoeba. (Note that some programming tools are described in greater detail in the *Programming Guide*.) It is expected that the manual pages will be used frequently.

Throughout the manual there are references of the form *prv*(L), *ack*(U) and *bullet*(A). These refer to manual pages at the back of each guide. The following table explains what each classification refers to and in which guide the manual page can be found.

Letter	Type	Location
(A)	Administrative Program	System Administration Guide
(H)	Include file	Programming Guide
(L)	Library routine	Programming Guide
(T)	Test Program	System Administration Guide
(U)	User Utility	User Guide

There is a full index at the end of the manual providing a permuted cross-reference to all the material. It gives references to both the manual pages and the introductory material which should provide enough information to clarify any difficulties in understanding.

2 Introduction

2.1. History

The Amoeba distributed operating system has been in development and use since 1981. Amoeba was originally designed and implemented at the Vrije Universiteit in Amsterdam, The Netherlands by Prof. Andrew S. Tanenbaum and two of his Ph.D. students, Sape Mullender and Robbert van Renesse. From 1986 until 1990 it was developed jointly there and at the Centre for Mathematics and Computer Science, also in Amsterdam. Since then development has continued at the Vrije Universiteit. It has passed through several versions, each experimenting with different file servers, network protocols and remote procedure call mechanisms. Although Amoeba has reached a point where it seems relatively stable, it is still undergoing change and so it is important to take note of the various warnings and advice about the proposed design changes for future releases.

2.2. The Amoeba Design Philosophy

Amoeba has been developed with several ideas in mind. The first is that computers are rapidly becoming cheaper and faster. In one decade we have progressed from many people sharing one computer to each person having their own computer. As computers continue to get cheaper it should be possible for each person to have many computers available for individual use. This is already the case to some extent.

The second relates to the widespread use and increasing performance of computer networks. The need for a uniform communication mechanism between computers that are both on a local network or on a wide-area network is already apparent for many applications.

What is needed is the ability to deal with physically distributed hardware while using logically centralized software. Amoeba allows the connection of large numbers of computers, on both local and wide-area networks, in a coherent way that is easy to use and understand.

The basic idea behind Amoeba is to provide the users with the illusion of a single powerful timesharing system, when in fact the system is implemented on a collection of machines, potentially distributed across several countries. The chief design goals were to build a distributed system that would be small, simple to use, scalable to large numbers of processors, have a degree of fault-tolerance, have very high performance, including the possibility for parallelism, and above all be usable in a way that is transparent to the users.

What is meant by *transparent* can best be illustrated by contrasting it with a network operating system, in which each machine retains its own identity. With a network operating system, each user logs into one specific machine: their home machine. When a program is started, it executes on the home machine, unless the user gives an explicit command to run it elsewhere. Similarly, files are local unless a remote file system is explicitly mounted or files are explicitly copied. In short, the user is clearly aware that multiple independent computers exist, and must deal with them explicitly.

By contrast, in a transparent distributed system, users effectively log into the system as a whole, and not to any specific machine. When a program is run, the system, not the user, decides the best place to run it. The user is not even aware of this choice. Finally, there is a single, system-wide object naming service. The files in a single directory may be located on

different machines possibly in different countries. There is no concept of file transfer, uploading or downloading from servers, or mounting remote file systems. The fact that a file is remote is not visible at all.

2.3. The System Architecture

The Amoeba architecture consists of four principal components, as shown in figure 2.1. First are the workstations, one per user, on which users can carry out editing and other tasks that require fast interactive response. The workstations are all diskless, and are primarily used as intelligent terminals that do window management, rather than as computers for running complex user programs. Currently Suns, IBM PC/AT clones and X terminals can be used as workstations.

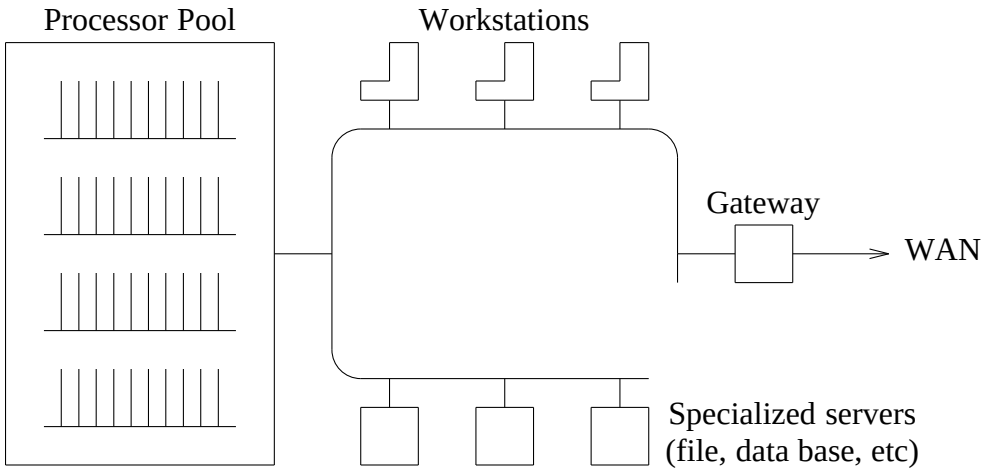


Fig. 2.1. The Amoeba architecture.

Second are the pool processors: a group of CPUs that can be dynamically allocated as needed, used, and then returned to the pool. This makes it possible to take advantage of parallelism within a job. For example, the *make* command might need to do six compilations, so six processors could be selected from the pool to do the compilation. Many applications, such as heuristic search in AI applications (e.g., playing chess), use large numbers of pool processors to do their computing. The processor pool also offers the possibility of doing many jobs in parallel (e.g., several large text processing jobs and program compilations) without affecting the perceived performance of the system because new work will be assigned to idle processors (or the most lightly loaded ones).

Third are the specialized servers, such as directory servers, file servers, boot servers, and various other servers with specialized functions. Each server is dedicated to performing a specific function. In some cases, there are multiple servers that provide the same function, for example, as part of the replicated file system.

Fourth are the gateways, which are used to link Amoeba systems at different sites and different countries into a single, uniform system. The gateways isolate Amoeba from the peculiarities of the protocols that must be used over the wide-area networks.

All the Amoeba machines run the same kernel, which primarily provides multithreaded processes, communication services, and little else. The basic idea behind the kernel was to keep it small, to enhance its reliability, and to allow as much as possible of the operating system to run as user processes (i.e., outside the kernel), providing for flexibility and experimentation. This approach is called a *microkernel*.

2.4. Objects and Capabilities

Amoeba is object-based. It uses capabilities for accessing objects. The Amoeba system can be viewed as a collection of objects, on each of which there is a set of operations that can be performed. For example, typical operations for a file object are creating, reading and deleting. The list of allowed operations is defined by the person who designs the object and who writes the code to implement it. Both hardware and software objects exist.

Associated with each object is a *capability*, a kind of ticket or key that allows the holder of the capability to perform some (not necessarily all) operations on that object. For example, a user process might have a capability for a file that permitted it to read the file, but not to modify it. Capabilities are protected cryptographically to prevent users from tampering with them.

Each user process owns some collection of capabilities, which together define the set of objects it may access and the type of operations it may perform on each. Thus capabilities provide a unified mechanism for naming, accessing and protecting objects. From the user's perspective, the function of the operating system is to create an environment in which objects can be created and manipulated in a protected way.

2.5. Remote Procedure Call

This object-based model visible to the users is *implemented* using remote procedure call. Associated with each object is a *server* process that manages the object. When a user process wants to perform an operation on an object, it sends a request message to the server that manages the object. The message contains the capability for the object, a specification of the operation to be performed, and any parameters required for the operation. The user, known as the *client*, then blocks. After the server has performed the operation, it sends back a reply message that unblocks the client. The combination of sending a request message, blocking, and accepting a reply message forms the remote procedure call, which can be encapsulated using stub routines, to make the entire remote operation look like a local procedure call.

A remote procedure call can impact perceived performance because it is possible for it to fail when it cannot find a server (temporarily or otherwise). Although this is a rare occurrence, it does mean that sometimes there may be a brief delay while a program attempts to locate a server. The delay may vary between two and thirty seconds. (Of course it is initially difficult to distinguish between a long timeout and someone rebooting the machine where your process is running, but that is a separate issue.)

2.6. Directories

Under Amoeba, file names are not implemented by the file server. There are many more object types than just files, so a more general naming service is required. The naming service is implemented by the directory server which implements an arbitrary directed graph of

directories. This graph can contain cycles and does so in the standard distribution. This is fully intentional and causes no problems. The directory object is a list of **(**ASCII string, capability-set**)** pairs. The *ASCII string* is the human readable name of the object and the *capability-set* contains one or more capabilities for the various copies of that object. (This provides support for fault tolerance by allowing the same name to refer to the replicated copies of an object.)

The capability for any object, including another directory, may be stored in a directory, so a directory can be linked into several places in the directory graph simply by entering its capability in several directories.

The directory graph has no concept of an absolute *root* since there is no tree structure. However, the user interface implements the root for a particular user as the user's login directory. Therefore all users have their own '/' directory.

An extra degree of fault tolerance is achieved by duplicating the directory server. Two copies of it can run simultaneously, each with its own set of data. The two servers communicate with each other to keep the data about directories consistent. If one server goes down, the other detects this and carries on functioning. When its partner is restarted it is brought up to date with the latest changes by the surviving server and they resume cooperation.

This provides a very high reliability in the directory service and allows software upgrades to the directory server without down time by simply stopping the servers one at a time and starting the new version.

2.7. Files

Files are also unusual in Amoeba. The file server delivered with Amoeba is known as the Bullet Server. It uses a very large buffer cache and stores files contiguously, both in core and on disk to give very high performance.

The semantics of Bullet Server files are different from conventional file systems. The foremost difference is that they are immutable. That is, once created a file cannot be changed. The only significant operations permitted on a file are read and delete. (There are some operations which deal with the status information of the file.)

The second aspect of files is that they are created atomically. They can be constructed in the Bullet Server's buffer cache but they do not officially exist until they are *committed*. At this point they are written to disk and may be used.

2.8. Amoeba and POSIX Emulation

To provide Amoeba with a reasonable programming environment and to simplify the migration of software and staff from UNIX-like systems to Amoeba, there is a POSIX emulation library which provides reasonable source code compatibility. A special server handles state information when necessary, but this is not explicitly visible to the user. It is started when required.

Functional equivalents of many of the standard POSIX tools and utilities are provided with Amoeba. In practice this means that the user environment of Amoeba is strikingly like that of POSIX, although no AT&T code is present. However the directory service and file semantics are significantly different, so it is important not to take the similarity with UNIX-

like systems for granted. For example, commands like *tail −f* are not implementable since file creation is atomic. Recursive commands such as *find*, *tar* and *rm −rf* are potentially dangerous if there are any cycles in the subgraph to which they are applied.

2.9. Conclusion

Amoeba was designed to provide a transparent distributed system. Although it is still undergoing further development the current version goes a long way towards satisfying the design goals.

The POSIX emulation has provided a ready-made programming environment for people to begin using Amoeba immediately. To help get started, the next section provides a tutorial introduction to using Amoeba and highlights some of the areas where Amoeba differs from POSIX.

3 Getting Started

This is a tutorial introduction for beginners which gives the basic information about how to log in and out, start up the window server, use some simple tools and navigate through the directory structure. The first section on logging in and out is intended to get you started quickly. It has been written with beginners in mind.

3.1. Logging In and Out

Before you can use Amoeba it is necessary to go to the system administrator and request a *user name*. This is the name by which you are known to the computer system. When allocated this name, resources will be set aside in the system for your use. This name is used to gain access to these and other system resources.

Amoeba uses passwords to prevent unauthorized use of users' accounts. When you are assigned a user name you must also give a password to go with the name. The password should be kept secret.

It is important to choose a password that will be difficult for others to guess. Names of wives, husbands, children, girlfriends or boyfriends are extremely bad choices of password. So are words from English or the local spoken language. Make sure, however, that the password is not too difficult to remember. There is a program to change your password and it should be used regularly to ensure security.

When you want to use Amoeba it will require you to give your user name and password before permitting you to access the system. This is known as *logging in*.

To use Amoeba you need a workstation. When you sit down at a free workstation you will see a short text followed by the word `Username` somewhere on the screen. For example:

```
Amoeba 5.3 login

Username:
```

The system is asking you to identify yourself by typing in the user name assigned to you by the system administrator. Therefore in response to this you should type in your user name and then press the key labeled *return*. In response to this the computer will print:

```
Password:
```

You should type in your password. The computer will not show it on the screen as it is typed. Once you have typed your password you should once again press the *return* key.

If you typed your password incorrectly it will print the message

```
Incorrect user name or password
```

and begin the login sequence again. If any other error message is printed then consult the system administrator.

If you typed your password correctly then a command interpreter will be started. A command interpreter is a program that accepts the commands typed at the keyboard and executes them. It will print a *prompt* to indicate that it is ready for you to enter commands. The prompt from the default command interpreter is

```
$
```

All the commands typed to the command interpreter are only executed after the *return* key is pressed. This is a signal to the computer that the whole command has been typed. In response to the prompt type the command

```
echo hello
```

Do not forget to press the *return* key after typing the two words. The system should respond by printing

```
hello
```

Another command to try is *date*. This should print Amoeba's idea of the current time.

As important as getting started and typing commands is knowing how to stop using the system. This is quite simple and is known as *logging out*. In response to the prompt

```
$
```

simply hold down the key labeled *control* and press the key for the character *D*. (It is not necessary to press the return key in this case!) The system may print a new line and then begin the login sequence.

3.2. Creating, Modifying and Destroying Files

As with most other computer systems, Amoeba provides files for the storage of data. A file is just an array of bytes. The file server distributed with Amoeba is called the Bullet Server. Users can create and modify files in many ways. Amoeba comes with several text editors for creating and modifying human readable text. Manual pages describe each editor and there is a section later in this manual containing the reference manuals for the editors. There are several editors currently available. One is called *elvis*(U). This is an imitation of the Berkeley *vi* editor, but provides some extra features as well. It has been linked to the names *vi* and *ex* under Amoeba. There is an Emacs-like editor called *jove*(U) and an editor called *ed*(U) which is very similar to the UNIX *ed* editor.

Files (and all other objects) are removed using the command *del*(U). To actually destroy an object it is necessary to use the **–f** flag of *del*. The **–f** flag should only be used if it is certain that nobody else has a link to the object or if further access to the information in it must be denied. At present there are no reference counts kept for files, so a garbage collection mechanism is used to destroy any files no longer referred to by any links.

The UNIX emulation also provides the *rm* command, but this does not destroy the file even with the **–f** flag. It simply removes the directory entry.

3.3. Customizing Your Environment

As with most systems, there are many aspects of the user's environment that can be customized. A very important choice for many people is that of command language interpreter. The command language interpreter is known as the *shell*. At present there are three shells available. They are the MINIX shell */bin/sh* (a clone of the *Bourne Shell*), the Korn Shell */bin/ksh*, and */bin/tcsh*. (The last is available as third-party software.) No doubt more shells will be ported to Amoeba in the future.

There are two types of environment variables under Amoeba. They are given to each process

by the creator of the process and after the process starts they can only be modified by the process itself. The first is the traditional string environment provided by many operating systems. This provides a mapping from a string (the environment variable name) to an arbitrary string. Many programs use this to have a fixed way of referring to a file by referring to the variable name. The variable in turn contains the name of the file to use. The second is the capability environment. This has a similar function to the string environment but maps the environment variable name to a capability. This is exactly the function of a directory (i.e., mapping a name to a capability), but is found inside the process' data and is not implemented with the semantics of a full directory server.

The simplest way to set up your initial environment is to modify the file called *Environ* in the directory ''/''. This file contains your encrypted password, initial shell and items to add to your capability and string environments. It is executed once when you log in and sets the initial environments for your shell.

The first line of the file contains your password. This you can set using the program *chpw*. The other lines of the file can be modified using a text editor. (Be careful not to modify the password line of the file.) The syntax for lines to set string environment variables is:

```
setenv VARIABLE value
```

The syntax to set capability environment variables is:

```
setcap VARIABLE pathname
```

where *pathname* is a path in the directory server which can be looked up to return a capability to insert into the environment. (NB. The path is not put into the environment!) The initial shell is specified by the line:

```
exec command
```

Because the shells need UNIX emulation, it is necessary to first start a *session*(U) server. The shell does not start it automatically. The following lines in the environment file will start the *session* server and the shell */bin/sh*:

```
setenv SHELL /bin/sh
setenv SPMASK 0xff:0x2:0x4
exec /profile/util/session -a /bin/sh
```

SPMASK is used to set the default column masks which control the directory entry protection. (For more information see the section below about directories and *spmask*(L).) It must be set in */Environ* so that the session server also knows the value. Note that if this variable is altered in the shell there is no way of exporting the value to the session server, which creates many files for the shell.

In addition to the *Environ* files, many shells also have their own start up file(s) for configuring the environment. The two shells provided with Amoeba use the file called *$HOME/.profile* where HOME is set by the *login*(A) program.

3.4. Starting X Windows

If the system being used has workstations with bitmapped displays, such as a SPARCstation or an X terminal, it is possible to use the X window system. The binaries for X windows are normally found in the directory */profile/module/x11/bin*. To use X windows, this directory must be in your path. To start X windows the following steps are required. If a workstation running Amoeba is being used, ensure that the workstation is running an Amoeba workstation kernel. This has the necessary drivers for X windows. Then it is necessary to start an X server on the workstation. The system administrator can arrange for this to happen automatically using the *boot*(A) server.

If an X terminal is to be used, then when it starts up it is assumed that it is already running an X server. It is necessary to have a TCP/IP server running on an Amoeba host when using an X terminal. The system administrator must arrange this.

The last step required is that the system administrator arrange that a login process (known as *xlogin*(A)) be started which communicates with the X server on the workstation. This will prompt for the user name and password in a similar manner to that previously described in the section on logging in.

Once logged in it is possible to start a window manager and create windows running on Amoeba. If there is a TCP/IP server available it is also possible to open windows where the processes run on a UNIX system. See the section *Remote Login* later in this manual for details.

The best place from which to start a window manager and some windows is in your *.profile* file. This is normally kept in your login directory. This is a command script executed by your shell when you log in. Figure figure 3.1 gives an example of a *.profile* file which can optionally start X windows.

The *.profile* calls the script *.mystartX* to start up X windows so that it is done in a separate process from the login session. This is necessary due to peculiarities with signals and X windows. The script *.mystartX* might be as in figure 3.2.

When you log in on a machine running an X server it is customary to start an *xterm* window. A shell will run in the xterm window and will execute your *.profile*. The first few lines of the *.profile* will set some important string environment variables. It is important to note that if the variable `TCP_SERVER` is not set, then you will use the default TCP/IP server. This may result in poor performance if this server is overloaded. The `PATH` environment variable tells the shell where to look for the commands that it should execute. The `PS1` variable describes what the primary command prompt should look like.

The *if* statement is checking to see if the workstation is running an X server. If not then it enquires whether the terminal type is a *xterm*. If it is just press return. (The type *xterm* should be replaced with the terminal most commonly used.) Otherwise fill in the name of the terminal type as described in the file */etc/termcap*. If unsure about this information consult the system administrator.

If there is an X server running then the terminal type is set to *xterm.* Then the X resources data base is constructed, the *twm* window manager started and then two simple X utilities are started.

For further details it is recommended that you consult the X windows documentation and/or an experienced X windows user.

```
PS1="Amoeba> "
PATH=.:/bin:/profile/util:/profile/module/x11/bin:
TCP_SERVER=/profile/cap/ipsvr/tcp
export PATH PS1 TCP_SERVER

if [ X$DISPLAY != X ]
then
          TERM=xterm
          export TERM
          .mystartX
else
          echo -n 'TERM=(xterm)'
          read answer
          case $answer in
          "") TERM=xterm;;
          *) TERM=$answer;;
          esac
          export TERM
fi
```

Fig. 3.1. An example *.profile* file

```
#!/bin/sh
xrdb -nocpp $HOME/.Xresources
twm &
sleep 4
xclock -geometry 80x80-2+2 -analog &
xload -geometry 140x80-88+2 &
xbiff -geometry 50x50-2+89 -file /home/mailbox &
```

Fig. 3.2. An example *.mystartX* script

3.5. Directories

The directory server provides a way of naming objects. It does this by mapping human readable strings to capabilities. The name used to reference an object is known as a *path name*. Under Amoeba your login directory has the name ‘‘/’’ (pronounced slash). All other directory names are ASCII strings not containing the character ‘‘/’’ and a path name is simply a list of names separated by slashes. All but the last name must be directory names. There are several programs for examining the directory structure and adding and deleting directories.

If a path name begins with ‘‘/’’ then it is absolute from your login directory. An example of an absolute path name is:

```
/profile/cap/bulletsvr/default
```

Using the shells provided with Amoeba it is possible to move about within the directory structure and the system will remember the current directory. If a path name does not begin with a ‘‘/’’ then it is interpreted as being relative to the current directory.

Unlike many other directory systems, the Amoeba directory server does not enforce a hierarchical tree structure. It allows an arbitrary graph with cycles and cross links. To assist in the navigation through the source tree there are two special directory names defined. The first is ‘‘.’’ (pronounced dot) and refers to the current directory. The second is ‘‘..’’ (pronounced dot dot) and refers to the previous directory in the current path name. The names ‘‘.’’ and ‘‘..’’ cannot be referred to as real directories but are interpreted by the directory path look-up software. Attempts to modify ‘‘.’’ or ‘‘..’’ entries will fail. (Note that this avoids the problem with the UNIX definition of ‘‘.’’ and ‘‘..’’ which do not return along symbolic links.)

3.5.1. The Soap Directory Server

The name (or directory) server distributed with Amoeba is called *Soap*. It maps ASCII names to capabilities. Unlike many other systems, the directory structure is not a tree. It is a arbitrary directed graph containing cycles.

3.5.2. The Directory Graph

The graph in figure 3.3 shows the standard directory structure for Amoeba, starting from the root directory for user *Tom*. (NB. there may be some variation from site to site.) This should provide some idea of how to find things. Remember that the root directory for each user is his login directory. Also note that the link */profile* restricts access rights so */profile/group/users/tom* will have fewer rights than the directory /, although they both refer to the same object.

The directories */bin*, */etc*, */lib* and */usr* contain various files necessary for the UNIX emulation.

3.5.3. Commands for Soap

For the user there are six basic commands for manipulating, and getting information from, directories:

dir list the contents of a directory.

del delete an entry in a directory and optionally the associated object.

mkd create a new directory.

chm change the mode of a directory entry.

get get the capability-set associated with a directory entry and write it on standard output.

put take a capability-set from standard input and create the named directory entry.

These commands are described in the manual pages of this volume. It is worth learning how they differ from the UNIX equivalents.

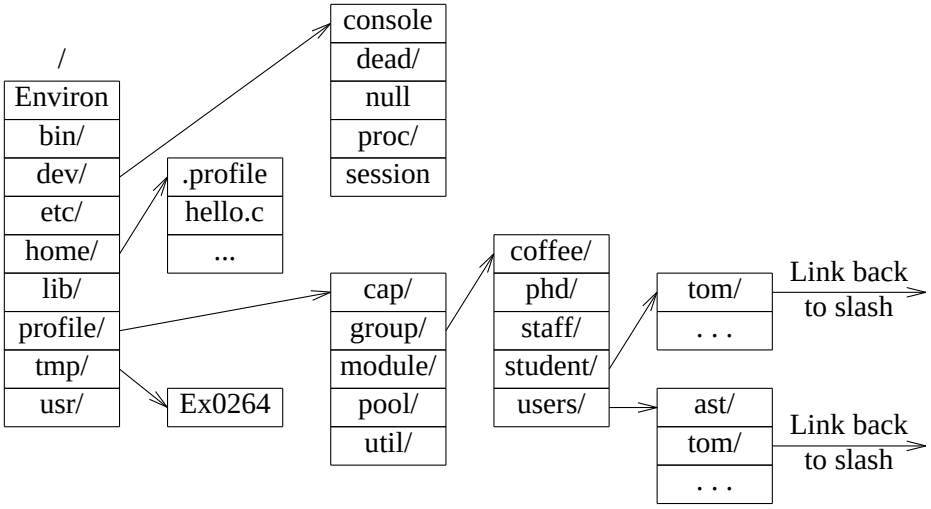


Fig. 3.3. The Directory Structure as seen by user *Tom*. Names that end with a / are directories.

3.5.4. The Structure of a Directory

A directory is a two-dimensional matrix. Each row and column is labeled by an ASCII string. Within a directory, all row and column names are unique. Also associated with each row is a set of capabilities. Each row maps its ASCII name to the set of capabilities in that row. The columns describe the access rights of the different groups that have access to the objects.

For example, a directory could be set up with three columns: the owner, the owner's group, and others, as in UNIX. The directory shown below handles the binding for two files, *README* and *x.c*, and a directory *profile*.

Name	Cap set	*Owner*	*Group*	*Other*
README	file 1	111	100	100
profile	directory	111	110	100
x.c	file 2	110	100	000

The owner of the directory has defined three access groups: the owner, group members, and others with whom the owner may share these objects. Each entry contains a set of three rights mask bits, for example, for read, write, and delete respectively. In our example, users that are in the same group as the owner of this directory are not allowed to write or delete the file *x.c*. Others are not allowed any access to this file. To prevent accidents, the owner has also turned off the delete right.

Although an object pointed to by the *owner*, *group* and *other* columns is one and the same object, the capabilities for them are not, because they have different access rights. Thus these capabilities will have different Rights and Check fields.

A capability for a directory specifies a particular column (or columns). For example, a

capability which is only good for *group* members will not return capabilities with all the rights bits on, but will restrict them according to the rights mask in the *group* column. The directory service can generate restricted capabilities for the various columns on-the-fly as needed, but it can also decide to cache them, at its discretion.

When a new version of an object is available, the owner presents the directory service with a new **(**name, capability**)** pair. If this is replacing an existing directory entry, all the capabilities in the directory entry are replaced with the new capability. Extra replicas of the new object may be created automatically by the object manager.

Operations on Directories

The bits in the *Rights Field* in a capability for a directory are defined as follows:

bits 0-3 These bits tell which columns in the directory may be accessed for look-up. Bit 0 is set if the first column may be accessed, bit 1 if the second column may be accessed, etc. to a maximum of four columns. The rights mask to be used for creating the capability in a look-up operation is formed by OR-ing together the masks in the accessible columns.

bits 4-5 These bits are reserved for future use.

bit 6 This bit gives the right to modify the directory, that is, adding, deleting and changing rows in the directory.

bit 7 This bit gives the right to delete a directory. Note that non-empty directories can be removed.

Note that other directory operations, such as listing the names of rows and columns, are available to any holder of a capability. They are not protected by rights bits.

3.6. Tools and Languages

Amoeba provides several programming tools and languages. Besides many of the standard tools from UNIX, such as *yacc*, there is a version management system for building programs and keeping them up to date. This program is called *amake*(U). For most uses, all that is necessary is to include the standard tools and provide a list of source files plus the name of the object to be made. All the dependency information is deduced and checked by *amake*.

Amoeba is distributed with compilers made using the Amsterdam Compiler Kit (ACK) which is described in some detail in the programming guide. It provides several languages and can generate code for many different kinds of machine. It can be used for cross-compiling when porting code to new systems. The programming languages C, Pascal, FORTRAN 77, BASIC and Modula 2 are provided. In addition assemblers are provided for several architectures. The GNU C compiler is available as third-party software. The Vrije Universiteit also provides an Amoeba version of the language Orca which was specifically designed for parallel programming. In addition there are several other programming tools. How to use all these tools is described briefly in the manual pages and in detail in the *Programming Guide*.

3.6.1. Debugging

At present there is very limited support for debugging programs. The GNU debugger is available as third-party software. It provides symbolic debugging and has support for threads.

There are tools for analyzing core dumps such as *nm*(U) and *pdump*(U). Note that to get core dumps it is necessary to make the directory */dev/dead.* Core dumps are stored here with as file name the process id of the process that dumped core. This number is printed when the core dump is made so that it is easy to find. It is important to clean out this directory from time to time to avoid filling up your disk.

Another tool which is useful for debugging suitably instrumented servers is the *monitor*(U) command. The server must be compiled with monitoring macros in it before this can be used.

There is also support for profiling user programs to look for performance bottlenecks. See *gprof*(U) and *profiling(L)* for further details.

3.7. On-line Help

The on-line help is provided by the command *aman*(U). It provides possibilities for keyword searches of the manuals as well as selecting and displaying manual pages.

4 Advanced Users

4.1. The Pool Processors

The Amoeba pool processors are usually allocated dynamically as needed. A special server (called the *run*(A) server) knows about the set of pool processors available to each user. It allocates processors for a user only from his pool. This consists of the processors in the subdirectories of */profile/pool*. The *run* server regularly collects information about the amount of free memory and the CPU load of each pool processor. It uses this to select the best processor to run a process on. If the *run* server has not been running then the processor selected will be on the basis of round-robin through the processors in */profile/pool*. This is particularly slow since the attempt to use the *run* server must time-out before it resorts to the round-robin strategy. If command startup seems very slow then the run server may not be present. If the run server crashed then it is not possibly for non-privileged users to execute processes until the run server is restarted.

It is possible to have more than one subdirectory in */profile/pool*. For each architecture in the pool there should be a subdirectory with the name of the architecture. If the run server is aware of all the pool (sub)directories, all the machines in all the subdirectories will form the processor pool for that user.

Since it is not normally necessary to know where processes are running the user is not told. The command *aps*(U) is used to find out which processes are running and what their status is. Privileged users can also find out where processes are running by using the **–s** option. The user name that *aps* prints with each process is merely a comment added when the process is started. It is not to be taken at face value since it is not really possible to determine the true owner.

While logged in, all users have a *session*(U) server. It provides support for pipes and process management. It also provides a way of keeping track of processes. It stores the capability of each process that the user starts in the directory */dev/proc*. To find out what processes you are running (that the session server knows about) it is sufficient to use the command

```
dir -l /dev/proc
```

It is possible to have orphan processes not attached to your session server. The command

```
aps -av
```

can be used to see if you have any orphan processes. The `OWNER` column of *aps* will show ‘‘no owner’’ or ‘‘not found’’ for orphan processes.

The command *stun*(U) is used to kill processes that are not accessible by the interrupt character from the owner’s terminal. It is much more powerful that the *kill*(U) command in the POSIX emulation. To kill a process it is sufficient to find out its process identifier using *aps*(U) or *dir*(U) and then the command

```
stun /dev/proc/pid
```

(where *pid* is the process identifier) will stun it. If this leaves the process hanging then the command

```
stun -s -1 /dev/proc/pid
```

will normally cause it to be cleaned up. If a process is an orphan then you may not have

sufficient rights to kill it. The system administrator may have to kill it for you.

4.2. Electronic Mail

The electronic mail system MMDF II is delivered with Amoeba. This provides the possibility to send and receive electronic mail. Mail messages are sent using the command *send* and incoming mail is read using the command *msg*. Before the mail system can be used to send mail the system administrator must configure the mail system. If electronic mail is to be sent to sites on the Internet then the TCP/IP server must be installed and configured and a physical network connection made to the Internet. Most university sites can simply connect Amoeba to the local UNIX network.

4.3. The POSIX Emulation

Amoeba provides versions of many of the utilities normally available under UNIX. These are normally found in */bin*. Native Amoeba utilities are normally found in */profile/util*. In some cases there are POSIX emulation utilities with the same functionality as native Amoeba utilities but they are provided for people familiar with UNIX to simplify getting started.

To implement things like shared file descriptors between processes, pipe-lines and */dev/tty* there is a server known as the session server (see *session*(U)). Usually this is started when your login shell is started. Only one is needed per user and the user is not normally aware of its presence. However, if a user starts a lot of processes that want to do a lot of I/O then the session server is a potential performance bottleneck.

At first glance Amoeba's POSIX emulation gives the illusion that you are using a truly UNIX-like system. It is exactly that. An illusion! There are several aspects of Amoeba which are subtly incompatible with the usual UNIX environment. It is important to understand these differences to avoid making costly and unpleasant mistakes. Most of these relate to incompatibilities due to the Soap Directory Server and the immutable files of the Bullet File Server. Note that the ASCII name of an object is not in the same server as the object. A server identifies objects using capabilities, not names. For example, files may be stored in the Bullet Server, while ASCII names for files are stored in the Soap Directory Server. Under UNIX the file service and the name service are tightly intertwined. This difference between UNIX and Amoeba is necessary since the name server provides names for more object types than just files.

One major difference to bear in mind is that the directory structure is an arbitrary directed graph with cycles. Therefore commands such as

```
rm -rf foo
```

are dangerous if the directory *foo* has a subgraph containing cycles, or worse yet contains a link to some point in the graph which leads back to *foo*.

The next major difference concerns links. There are no symbolic links in the Soap Directory Server. The only kind of links are hard links but they have different semantics from those of POSIX hard links. The directory server stores **(**name, capability-set**)** pairs. This means that if you store the capability for an object under a the name *jack* and then run the command

```
ln jack jill
```

then the directory server stores the capability in two places. It does not keep link counts. If the capability *jack* is now replaced with a new capability (because somebody made a new

version of the object *jack*), the entry for *jill* will still have the original capability that it stored when the *ln* command was run. Thus the link is broken. These semantics are important to bear in mind when deleting objects. If the command

```
del -f jack
```

is executed it will tell the server to destroy the object *jack*. The server will simply comply and destroy the object (assuming that the capability has sufficient rights), regardless of whether or not anyone else holds the capability for this object. There are no link counts anywhere so the object vanishes. Any capabilities still referring to the object will be invalid.

The Bullet Server implements immutable files. That is, once a Bullet file has been created it cannot be changed. If a file is edited, then the file is first copied, then modified and an entirely new file is created. The new file has a different capability from the original. This new capability is normally stored under the same name as the original file and the original file is deleted. For the reasons given above, this results in breaking any links to a file whenever it is modified. Note that directories are mutable. Changing a directory's contents does not change the capability for the directory.

These differences will normally not be a problem unless the POSIX semantics for links are required. One way to achieve these semantics is to make a directory in which the capability for the object is stored. Then make a link to the directory. Now if the capability for the object is changed, the links to the directory will not have changed and so the change will appear in the linked directories.

Since capabilities are used for protection of objects, Amoeba does not have or need the POSIX equivalent of user ids or group ids. The emulation library returns group id 1 and user id based on the object number in the capability for the directory /. See *posix*(L) for more details.

POSIX only knows about one type of object, namely the *file*. Since Amoeba knows about many more types of objects, programs such as *ls*(U) can get confused about what protection modes and file types to use for some objects, since they try to interpret everything as a file or directory. In particular, if a capability for an object that no longer exists, or whose server is down, is stored in a directory then *ls* will sometimes describe this directory entry as being for a character special device. Use the native Amoeba command *dir*(U) to get accurate information about the state of the objects in a directory

In general these differences between POSIX and Amoeba should not present many problems as long as they are kept in mind.

5 Accessing Non-Amoeba Systems

5.1. Remote Login

It is possible to access non-Amoeba systems from Amoeba. For example, it is possible to execute programs on UNIX systems reachable from the Amoeba network. Amoeba provides a special server, called the TCP/IP server, which can be used to send messages to any hosts that can communicate using the TCP/IP network protocol. A special program, *ttn*(U) (a telnet implementation) provides a way of logging in to remote machines.

Amoeba's ability to use the TCP/IP network protocol also gives it the possibility of talking over wide-area networks, such as the Internet, and to communicate with other systems, including remote Amoeba LANs. This means it is possible to send electronic mail or use FTP. The electronic mail system MMDF has been ported to Amoeba which allows Amoeba to send and receive electronic mail to or from any site on the Internet.

5.2. File Transfer

It is possible to transfer files quickly between Amoeba and remote systems without using the *rcp* command or some equivalent on the remote host. The commands to do this are called *tob*(U) and *fromb*(U) Both these programs run on the remote host and require that the remote system have the Amoeba network driver built in. At present drivers are provided for various versions of UNIX. *Tob* stands for *to the bullet server* and can send a file from a remote machine to the Amoeba Bullet Server. Similarly *fromb* reads a file from the Amoeba Bullet Server and writes it on *stdout* on the remote machine. See the manual pages for these two commands for more details about how to use these facilities.

6 Editors and Text Processing

There are several editors available under Amoeba. In particular there is the *vi* clone *elvis*. This editor actually provides more functionality than *vi/ex* and the user-interface is subtly different, but the differences should not affect many people. There is also an *ed*(U) clone available in the third-party software collection.

The screen editor *jove*(U) is provided. It is patterned after *Emacs*, but is simpler. The program *teachjove*(U) is available for those wishing to learn how to use *jove*(U).

A version of the stream editor sed (see *sed*(U)) and awk (see *bawk*(U)) have also been provided. The GNU version of awk is available in the third-party software and is superior to bawk. It is recommended that this be installed. See the *Tools* section of the *Programming Guide* and the manual pages in this volume for further information.

Amoeba currently supports only one text formatter, namely TEX. This is provided free with Amoeba along with some slight modifications to make it run under Amoeba. It requires a large amount of memory to function well. The small TEX version requires about 4 or 5 Mbytes of memory to run, depending on the size of the input. The full TEX requires at least 8 Mbytes for moderately sized jobs. For full details on how to use it read the documentation provided with the TEX distribution.

6.1. Elvis Reference Manual

By Steve Kirkendall (E-Mail: kirkenda@cs.pdx.edu)

6.1.1. INTRODUCTION

Elvis is a clone of *vi/ex*, the standard UNIX editor. *Elvis* supports nearly all of the *vi/ex* commands, in both *visual mode* and *colon mode*.

Like *vi/ex*, *elvis* stores most of the text in a temporary file, instead of RAM. This allows it to edit files that are too large to fit in a single process' data space.

Elvis runs under Amoeba, BSD UNIX, AT&T SysV UNIX, Minix, MS-DOS, Atari TOS, Coherent, and OS9/68000. The next version is expected to add OS/2, VMS, AmigaDos, and MacOS. Contact the author before you start porting it to some other OS, because somebody else may have already done it.

Elvis is freely redistributable, in either source form or executable form. There are no restrictions on how you may use it.

Compiling

See the ''Versions'' section of this manual for instructions on how to compile *elvis*.

If you want to port *elvis* to another OS or compiler, then you should read the ''Portability'' part of the ''Internal'' section.

Overview of Elvis

The user interface of *elvis/vi/ex* is weird. There are two major command modes in *elvis*, and a few text input modes as well. Each command mode has a command which allows you to switch to the other mode.

You will probably use the *visual mode* most of the time. This is the mode that *elvis* normally starts up in.

In visual command mode, the entire screen is filled with lines of text from your file. Each keystroke is interpreted as part of a visual command. If you start typing text, it will **not** be inserted, it will be treated as part of a command. To insert text, you must first give an ''insert text'' command. This may take some getting used to. (An alternative exists. Lookup the ''inputmode'' option.)

The *colon mode* is quite different. *Elvis* displays a '':'' character on the bottom line of the screen, as a prompt. You are then expected to type in a command line and hit the <Return> key. The set of commands recognized in the colon mode is different from visual mode's.

The rest of this manual describes the commands available under each command mode followed by a description of the many options available. Thereafter comes a lot of information about the internal workings of *elvis* and other details which are mainly relevant to those compiling and installing *elvis*.

6.1.2. VISUAL MODE COMMANDS

Most visual mode commands are one keystroke long. The following table lists the operation performed by each keystroke, and also denotes any options or arguments that it accepts. Notes at the end of the table describe the notation used in this table.

In addition to the keys listed here, your keyboard's ''arrow'' keys will be interpreted as the appropriate cursor movement commands. The same goes for <PgUp> and <PgDn>, if your keyboard has them. There is a colon mode command (to be described later) which will allow you to define other keys, such as function keys.

A tip: visual command mode looks a lot like text input mode. If you forget which mode you are in, just hit the <Esc> key. If *elvis* beeps, then you are in visual command mode. If *elvis* does not beep, then you were in input mode, but by hitting <Esc> you will have switched to visual command mode. So, one way or another, after <Esc> *elvis* will be ready for a command.

command			description	type
	^A		—	
	^B		Move toward the top of the file by 1 screen full	
	^C		—	
count	^D		scroll down <count> lines (default 1/2 screen)	
count	^E		scroll up <count> lines	
	^F		move toward the bottom of the file by 1 screen full	
	^G		show file status, and the current line #	
count	^H		move left, like h	MOVE
	^I		—	
count	^J		move down	MOVE
	^K		—	
	^L		redraw the screen	
count	^M		move to the front of the next line	MOVE
count	^N		move down	MOVE
	^O		—	
count	^P		move up	MOVE
	^Q		—	
	^R		redraw the screen	
	^S		—	
	^T		—	
count	^U		scroll up <count> lines (default 1/2 screen)	
	^V		—	
	^W		—	
	^X		—	
count	^Y		scroll down <count> lines	
	^Z		—	
	ESC		—	
	^\		—	
	^]		if the cursor is on a tag name, go to that tag	
	^^		switch to the previous file, like ''**:e #**''	
	^_		—	
count	SPC		move right,like ''l''	MOVE
	!	mv	run the selected lines through an external filter program	
	"	key	select which cut buffer to use next	
count	#	+	increment a number	EDIT
	$		move to the rear of the current line	MOVE

	%		move to the matching (){}[] character	MOVE
count	&		repeat the previous ‘‘:s//’’ command here	EDIT
	’	key	move to a marked line	MOVE
count	(		move backward <count> sentences	MOVE
count	)		move forward <count> sentences	MOVE
	*		go to the next error in the errlist	
count	+		move to the front of the next line	MOVE
count	,		repeat the previous [fFtT] but in the other direction	MOVE
count	–		move to the front of the preceding line	MOVE
count	.		repeat the previous ‘‘edit’’ command	
	/text		search forward for a given regular expression	MOVE
	0		if not part of count, move to first char of this line	MOVE
	1		part of count	
	2		part of count	
	3		part of count	
	4		part of count	
	5		part of count	
	6		part of count	
	7		part of count	
	8		part of count	
	9		part of count	
	:	text	run single *ex* cmd	
count	;		repeat the previous [fFtT] cmd	MOVE
	<	mv	shift text left	EDIT
	=		—	
	>	mv	shift text right	EDIT
	?	text	search backward for a given regular expression	MOVE
	@	key	execute the contents of a cut-buffer as *vi* commands	
count	A	inp	append at end of the line	EDIT
count	B		move back Word	MOVE
	C	inp	change text from the cursor through the end of the line	EDIT
	D		delete text from the cursor through the end of the line	EDIT
count	E		move end of Word	MOVE
count	F	key	move leftward to a given character	MOVE
count	G		move to line #<count> (default is the bottom line)	MOVE
count	H		move to home row (the line at the top of the screen)	
count	I	inp	insert at the front of the line (after indents)	EDIT
count	J		join lines, to form one big line	EDIT
	K		look up keyword	
count	L		move to last row (the line at the bottom of the screen)	
	M		move to middle row	
	N		repeat previous search, but in the opposite direction	MOVE
count	O	inp	open up a new line above the current line	EDIT
	P		paste text before the cursor	
	Q		quit to *ex* mode	
	R	inp	overtype	EDIT
count	S	inp	change lines, like <count>cc	

count	T	key	move leftward *almost* to a given character	MOVE
	U		Undo all recent changes to the current line	
	V		—	
count	W		move forward <count> Words	MOVE
count	X		delete the character(s) to the left of the cursor	EDIT
count	Y		yank text line(s) (copy them into a cut buffer)	
	ZZ		save the file & exit	
	[[		move back 1 section	MOVE
	\		—	
	]]		move forward 1 section	MOVE
	^		move to the front of the current line (after indent)	MOVE
	_		—	
	‘		key move to a marked character	MOVE
count	a	inp	insert text after the cursor	EDIT
count	b		move back <count> words	MOVE
	c	mv	change text	EDIT
	d	mv	delete text	EDIT
count	e		move forward to the end of the current word	MOVE
count	f	key	move rightward to a given character	MOVE
	g		—	
count	h		move left	MOVE
count	i	inp	insert text at the cursor	EDIT
count	j		move down	MOVE
count	k		move up	MOVE
count	l		move right	MOVE
	m	key	mark a line or character	
	n		repeat the previous search	MOVE
count	o	inp	open a new line below the current line	EDIT
	p		paste text after the cursor	
	q		—	
count	r	key	replace <count> chars by a given character	EDIT
count	s	inp	replace <count> chars with text from the user	EDIT
count	t	key	move rightward *almost* to a given character	MOVE
	u		undo the previous edit command	
	v		—	
count	w		move forward <count> words	MOVE
count	x		delete the character that the cursor’s on	EDIT
	y	mv	yank text (copy it into a cut buffer)	
	z	key	scroll current line to the screen’s +=top –=bottom .=middle	
count	{		move back <count> paragraphs	MOVE
count	|		move to column <count> (the leftmost column is 1)	
count	}		move forward <count> paragraphs	MOVE
count	~		switch a character between uppercase & lowercase	EDIT
	DEL		—	

count

Many commands may be preceded by a count. This is a sequence of digits representing

a decimal number. For most commands that use a count, the command is repeated <count> times. The count is always optional, and usually defaults to 1.

key
: Some commands require two keystrokes. The first key always determines which command is to be executed. The second key is used as a parameter to the command.

mv
: Six commands (! < > c d y) operate on text between the cursor and some other position. To specify that other position, you are expected to follow the command keystroke with a movement command. Also, you may have the command operate on the whole line that the cursor is on by typing the command key a second time.

inp
: Many commands allow the user to interactively enter text.

EDIT
: These commands affect text, and may be repeated by the ‘‘.’’ command.

MOVE
: These commands move the cursor, and may be used to specify the extent of a member of the ‘‘mv’’ class of commands.

Input Mode

You cannot type text into your file directly from visual command mode. Instead, you must first give a command which will put you into input mode. The commands to do this are A/C/I/O/R/S/a/i/o/s.

The S/s/C/c commands temporarily place a $ at the end of the text that they are going to change.

In input mode, all keystrokes are inserted into the text at the cursor’s position, except for the following:

^A	insert a copy of the last input text
^D	delete one indent character
^H	(backspace) erase the character before the cursor
^L	redraw the screen
^M	(carriage return) insert a newline (^J, linefeed)
^P	insert the contents of the cut buffer
^R	redraw the screen, like ^L
^T	insert an indent character
^U	backspace to the beginning of the line
^V	insert the following keystroke, even if special
^W	backspace to the beginning of the current word
^Z^Z	write the file & exit *elvis*
^[	(ESCape) exit from input mode, back to command mode

Also, on some systems, ^S may stop output, ^Q may restart output, and ^C may interrupt execution. ^@ (the NUL character) cannot be inserted.

The R visual command puts you in overtype mode, which is a slightly different form of input

mode. In overtype mode, each time you insert a character, one of the old characters is deleted from the file.

Arrow keys in Input Mode

The arrow keys can be used to move the cursor in input mode. (This is an extension; the real *vi* does not support arrow keys in input mode.) The <PgUp>, <PgDn>, <Home>, and <End> keys work in input mode, too. The <Delete> key deletes a single character in input mode.

The best thing about allowing arrow keys to work in input mode is that as long as you are in input mode, *elvis* seems to have a fairly ordinary user interface. With most other text editors, you are always in either insert mode or replace mode, and you can use the arrow keys at any time to move the cursor. Now, *elvis* can act like that, too. In fact, with the new ‘‘inputmode’’ option and the ‘‘CTRL−Z CTRL−Z’’ input command, you may never have to go into visual command mode for simple edit sessions.

Digraphs

Elvis supports digraphs as a way to enter non-ASCII characters. A digraph is a character which is composed of two other characters. For example, an apostrophe and the letter i could be defined as a digraph which is to be stored and displayed as an accented i.

There is no single standard for extended ASCII character sets. *Elvis* can be compiled to fill the digraph with values appropriate for either the IBM PC character set, or the LATIN−1 character set used by X windows, or neither. (See the discussions of `−DCS_IBMPC` and `−DCS_LATIN1` in the CFLAGS section of this manual.) You can view or edit the digraph table via the ‘‘:digraph’’ colon command.

Digraphs wil not be recognized until you have entered ‘‘:set digraph’’.

To actually use a digraph type the first character, then hit <Backspace>, and then type the second character. *Elvis* will then substitute the non-ASCII character in their place.

Abbreviations

Elvis can expand abbreviations for you. You define an abbreviation with the `:abbr` command, and then whenever you type in the abbreviated form while in input mode, *elvis* will immediately use the long form.

Elvis does not perform the substitution until you type a non-alphanumeric character to mark the end of the word. If you type a CTRL−V before that non-alphanumeric character, then *elvis* will not perform the substitution.

Auto-Indent

With the ‘‘:set autoindent’’ option turned on, *elvis* will automatically insert leading white space at the beginning of each new line that you type in. The leading white space is copied from the preceding line.

To add more leading white space, type CTRL−T. To remove some white space, type

CTRL–D.

Elvis' autoindent mode is not 100% compatible with *vi*'s. In *elvis*, 0^D and ^^D do not work, ^U can wipe out all indentation, and sometimes *elvis* will use a different amount of indentation than *vi* would.

6.1.3. COLON MODE COMMANDS

lines	command	arguments
[line]	**a**ppends	
	args	[files]
	cc	[files]
	cd	[directory]
[line][,line]	**c**hange	
	chdir	[directory]
[line][,line]	**co**py	line
[line][,line]	**d**elete	["x]
	digraph[!]	[XX [Y]]
	edit[!]	[file]
	errlist[!]	[errlist]
	ex[!]	[file]
	file	[file]
[line][,line]	**g**lobal	/regexp/ command
[line]	**i**nsert	
[line][,line]	**j**oin	
[line][,line]	**l**ist	
	make	[target]
	map[!]	key mapped_to
[line]	mar**k**	x
	mkexrc	
[line][,line]	**m**ove	line
	next[!]	[files]
	Next[!]	
[line][,line]	**nu**mber	
	previous[!]	
[line][,line]	**p**rint	
[line]	**pu**t	["x]
	quit[!]	
[line]	**r**ead	file
	rewind[!]	
	set	[options]
	source	file
[line][,line]	**s**ubstitute	/regexp/replacement/[p][g][c]
	tag[!]	tagname
[line][,line]	**t**o	line
	undo	
	unmap[!]	ey

	version	
[line][,line]	**v**global	regexp/ command
	visual	
	wq	
[line][,line]	**w**rite[!]	[[>>]file]
	xit[!]	
[line][,line]	**y**ank	["x]
[line][,line]	**!**	command
[line][,line]	<	
[line][,line]	=	
[line][,line]	>	
[line][,line]	**&**	
	@	"x

To use colon mode commands, you must switch from visual command mode to colon command mode. The visual mode commands to do this are ‘‘:’’ for a single colon command, or ‘‘Q’’ for many colon mode commands.

Line Specifiers

Line specifiers are always optional. The first line specifier of most commands usually defaults to the current line. The second line specifier usually defaults to be the same as the first line specifier. Exceptions are :write, :global, and :vglobal, which act on all lines of the file by default, and :!, which acts on no lines by default.

Line specifiers consist of an absolute part and a relative part. The absolute part of a line specifier may be either an explicit line number, a mark, a dot to denote the current line, a dollar sign to denote the last line of the file, or a forward or backward search.

An explicit line number is simply a decimal number, expressed as a string of digits.

A mark is typed in as an apostrophe followed by a letter. Marks must be set before they can be used. You can set a mark in visual command mode by typing ‘‘m’’ and a letter, or you can set it in colon command mode via the ‘‘mark’’ command.

A forward search is typed in as a regular expression surrounded by slash characters; searching begins at the default line. A backward search is typed in as a regular expression surrounded by question marks; searching begins at the line before the default line.

If you omit the absolute part, then the default line is used.

The relative part of a line specifier is typed as a ‘‘+’’ or ‘‘–’’ character followed by a decimal number. The number is added to or subtracted from the absolute part of the line specifier to produce the final line number.

As a special case, the % character may be used to specify all lines of the file. It is roughly equivalent to saying 1,$. This can be a handy shortcut.

Some examples:

:p	print the current line
:37p	print line 37
:’gp	print the line which contains mark g

:/foo/p	print the next line that contains ‘‘foo’’
:$p	print the last line of the file
:20,30p	print lines 20 through 30
:1,$p	print all lines of the file
:%p	print all lines of the file
:/foo/–2,+4p	print 5 lines around the next ‘‘foo’’

Text Entry Commands

```
[line] append
[line][,line] change ["x]
[line] insert
```

The **a**ppend command inserts text after the specified line.

The **i**nsert command inserts text before the specified line.

The **c**hange command copies the range of lines into a cut buffer, deletes them, and inserts new text where the old text used to be.

For all of these commands, you indicate the end of the text you are inserting by hitting ^D or by entering a line which contains only a period.

Cut & Paste Commands

```
[line][,line] delete ["x]
[line][,line] yank ["x]
[line] put ["x]
[line][,line] copy line
[line][,line] to line
[line][,line] move line
```

The **d**elete command copies the specified range of lines into a cut buffer, and then deletes them.

The **y**ank command copies the specified range of lines into a cut buffer, but does *not* delete them.

The **pu**t command inserts text from a cut buffer after the specified line.

The **co**py and **t**o commands yank the specified range of lines and then immediately paste them after some other line.

The **m**ove command deletes the specified range of lines and then immediately pastes them after some other line. If the destination line comes after the deleted text, then it will be adjusted automatically to account for the deleted lines.

Display Text Commands

```
[line][,line] print
[line][,line] list
[line][,line] number
```

The **p**rint command displays the specified range of lines.

The **nu**mber command displays the lines, with line numbers.

The **l**ist command also displays them, but it is careful to make control characters visible.

Global Operations Commands

```
[line][,line] global /regexp/ command
[line][,line] vglobal /regexp/ command
```

The **g**lobal command searches through the lines of the specified range (or through the whole file if no range is specified) for lines that contain a given regular expression. It then moves the cursor to each of these lines and runs some other command on them.

The **v**global command is similar, but it searches for lines that **do not** contain the regular expression.

Line Editing Commands

```
[line][,line] join
[line][,line] ! program
[line][,line] <
[line][,line] >
[line][,line] substitute /regexp/replacement/[p][g][c]
[line][,line] &
```

The **j**oin command catenates all lines in the specified range together to form one big line. If only a single line is specified, then the following line is catenated onto it.

The **!** command runs an external filter program, and feeds the specified range of lines to its *stdin*. The lines are then replaced by the output of the filter. A typical example would be ‘‘:’a,’z!sort’’ to sort the lines ’a,’z.

The < and > commands shift the specified range of lines left or right, normally by the width of 1 tab character. The ‘‘shiftwidth’’ option determines the shifting amount.

The **s**ubstitute command finds the regular expression in each line, and replaces it with the replacement text. The ‘‘p’’ option causes the altered lines to be printed. The ‘‘g’’ option permits all instances of the regular expression to be found & replaced. (Without ‘‘g’’, only the first occurrence in each line is replaced.) The ‘‘c’’ option asks for confirmation before each substitution.

The **&** command repeats the previous substitution command. Actually, ‘‘&’’ is equivalent to ‘‘s//~/’’ with the same options as last time. It searches for the last regular expression that you specified for any purpose, and replaces it with the same text that was used in the previous

substitution.

Undo Command

```
undo
```

The **u**ndo command restores the file to the state it was in before your most recent command which changed text.

Configuration & Status Commands

```
map[!] [key mapped_to]
unmap[!] key
abbr [word expanded_form_of_word]
unabbr word
digraph[!] [XX [Y]]
set [options]
mkexrc
[line] mark "x
visual
version
[line][,line] =
file [file]
source file
@ "x
```

The **ma**p command allows you to configure *elvis* to recognize your function keys, and treat them as though they transmitted some other sequence of characters. Normally this mapping is done only when in the visual command mode, but with the [!] present it will map keys under all contexts. When this command is given with no arguments, it prints a table showing all mappings currently in effect. When called with two arguments, the first is the sequence that your function key really sends, and the second is the sequence that you want *elvis* to treat it as having sent.

The **unm**ap command removes key definitions that were made via the map command.

The **ab**br command is used to define/list a table of abbreviations. The table contains both the abbreviated form and the fully spelled-out form. When you are in visual input mode, and you type in the abbreviated form, *elvis* will replace the abbreviated form with the fully spelled-out form. When this command is called without arguments, it lists the table; with two or more arguments, the first argument is taken as the abbreviated form, and the rest of the command line is the fully spelled-out form.

The **una**bbr command deletes entries from the abbr table.

The **di**graph command allows you to display the set of digraphs that *elvis* is using, or add/remove a digraph. To list the set of digraphs, use the digraph command with no arguments. To add a digraph, you should give the digraph command two arguments. The first argument is the two ASCII characters that are to be combined; the second is the non-

ASCII character that they represent. The non-ASCII character's most significant bit is automatically set by the digraph command, unless you append a ''!'' to the command name.

Removal of a digraph is similar to adding a digraph, except that you should leave off the second argument.

The **se**t command allows you examine or set various options. With no arguments, it displays the values of options that have been changed. With the single argument ''all'' it displays the values of all options, regardless of whether they have been explicitly set or not. Otherwise, the arguments are treated as options to be set.

The **mk**exrc command saves the current configuration to a file called *.exrc* in the current directory.

The mar**k** command defines a named mark to refer to a specific place in the file. This mark may be used later to specify lines for other commands.

The **vi**sual command puts the editor into visual mode. Instead of emulating *ex*, *elvis* will start emulating *vi*.

The **ve**rsion command tells you that what version of *elvis* this is.

The **=** command tells you what line you specified, or, if you specified a range of lines, it will tell you both endpoints and the number of lines included in the range.

The **f**ile command tells you the name of the file, whether it has been modified, the number of lines in the file, and the current line number. You can also use it to change the name of the current file.

The **so**urce command reads a sequence of colon mode commands from a file, and interprets them.

The **@** command executes the contents of a cut-buffer as *ex* commands.

Multiple File Commands

```
args [files]
next[!] [files]
Next[!]
previous[!]
rewind[!]
```

When you invoke *elvis* from your shell's command line, any file names that you give to *elvis* as arguments are stored in the args list. The **ar**gs command will display this list, or define a new one.

The **n**ext command switches from the current file to the next one in the args list. You may specify a new args list here, too.

The **N**ext and **pre**vious commands (they are really aliases for the same command) switch from the current file to the preceding file in the args list.

The **rew**ind command switches from the current file to the first file in the args list.

Switching Files

```
edit[!] [file]
tag[!] tagname
```

The **e**dit command allows to switch from the current file to some other file. This has nothing to do with the args list, by the way.

The **ta**g command looks up a given tagname in a file called ‘‘tags’’. This tells it which file the tag is in, and how to find it in that file. *Elvis* then switches to the tag’s file and finds the tag.

Working with a Compiler

```
cc [files]
make [target]
errlist[!] [errlist]
```

The **cc** and **mak**e commands execute your compiler or *make* utility and redirect any error messages into a file called *errlist*. By default, *cc* is run on the current file. (You should write it out before running *cc*.) The contents of the *errlist* file are then scanned for error messages. If an error message is found, then the cursor is moved to the line where the error was detected, and the description of the error is displayed on the status line.

After you have fixed one error, the **er**rlist command will move the cursor to the next error. In visual command mode, hitting ‘‘*’’ will do this, too.

You can also create an *errlist* file from outside *elvis* and use

```
elvis -m
```

to start *elvis* and have the cursor moved to the first error. Note that you do not need to supply a file name with

```
elvis -m
```

because the error messages always say which source file an error is in.

Note: When you use *errlist* repeatedly to fix several errors in a single file, it will attempt to adjust the reported line numbers to allow for lines that you have inserted or deleted. These adjustments are made with the assumption that you will work though the file from the beginning to the end.

Exit Commands

```
quit[!]
wq
xit
```

The **q**uit command exits from the editor without saving your file.

The **wq** command writes your file out, then exits.

The **x**it command is similar to the **wq** command, except that **x**it will not bother to write your file if you have not modified it.

File I/O Commands

```
[line] read file
[line][,line] write[!] [[>>]file]
```

The **r**ead command gets text from another file and inserts it after the specified line. It can also read the output of a program; simply precede the program name by a ‘‘!’’ and use it in place of the file name.

The **w**rite command writes the whole file, or just part of it, to some other file. The ‘‘!’’, if present, will permit the lines to be written even if you have set the readonly option. If you precede the file name by >> then the lines will be appended to the file.

Directory Commands

```
cd [directory]
chdir [directory]
shell
```

The **cd** and **chd**ir commands (really two names for one command) switch the current working directory.

The **sh**ell command starts an interactive shell.

Debugging Commands

```
[line][,line] debug[!]
validate[!]
```

These commands are only available if you compile *elvis* with the `-DDEBUG` flag.

The de**b**ug command lists statistics for the blocks which contain the specified range of lines. If the ‘‘!’’ is present, then the contents of those blocks is displayed, too.

The **va**lidate command checks certain variables for internal consistency. Normally it does not output anything unless it detects a problem. With the ‘‘!’’, though, it will always produce *some* output.

6.1.4. REGULAR EXPRESSIONS

Elvis uses regular expressions for searching and substitutions.

Syntax

Elvis' regexp package treats the following one or two character strings (called meta-characters) in special ways:

\(\)	Used to delimit subexpressions
^	Matches the beginning of a line
$	Matches the end of a line
\<	Matches the beginning of a word
\>	Matches the end of a word
[ ]	Matches any single character inside the brackets
*	The preceding may be repeated 0 or more times
\+	The preceding may be repeated 1 or more times
\?	The preceding is optional

Anything else is treated as a normal character which must match exactly. The special strings may all be preceded by a backslash to force them to be treated normally.

Options

Elvis has two options which affect the way regular expressions are used. These options may be examined or set via the :set command.

The first option is called *[no]magic*. This is a boolean option, and it is *magic* (TRUE) by default. While in magic mode, all of the meta-characters behave as described above. In *nomagic* mode, only ^ and $ retain their special meaning.

The second option is called *[no]ignorecase*. This is a boolean option, and it is *noignorecase* (FALSE) by default. While in *ignorecase* mode, the searching mechanism will not distinguish between an uppercase letter and its lowercase form. In *noignorecase* mode, uppercase and lowercase are treated as being different.

Also, the *[no]wrapscan* option affects searches.

Substitutions

The :s command has at least two arguments: a regular expression, and a substitution string. The text that matched the regular expression is replaced by text which is derived from the substitution string.

Most characters in the substitution string are copied into the text literally but a few have special meaning:

&	Insert a copy of the original text
~	Insert a copy of the previous replacement text
\1	Insert a copy of that portion of the original text which matched the first set of \(\) parentheses.
\2 – \9	Does the same for the second (etc.) pair of \(\).
\U	Convert all chars of any later &, ~, or \# to uppercase
\L	Convert all chars of any later &, ~, or \# to lowercase
\E	End the effect of \U or \L

\u	Convert the first char of the next &, ~ or \# to uppercase
\l	Convert the first char of the next &, ~ or \# to lowercase

These may be preceded by a backslash to force them to be treated normally. If *nomagic* mode is in effect, then & and ~ will be treated normally, and you must write them as \& and \~ for them to have special meaning.

Examples

This example changes every occurrence of ‘‘utilize’’ to ‘‘use’’:

```
:%s/utilize/use/g
```

The next example deletes all white space that occurs at the end of a line anywhere in the file. (The brackets contain a single space and a single tab.):

```
:%s/[ ]\+$//
```

The next example converts the current line to uppercase:

```
:s/.*/\U&/
```

This example underlines each letter in the current line, by changing it into an ‘‘underscore backspace letter’’ sequence. (The ^H is entered as ‘‘CTRL–V backspace’’.):

```
:s/[a-zA-Z]/_^H&/g
```

This example locates the last colon in a line, and swaps the text before the colon with the text after the colon. The first \(\) pair is used to delineate the stuff before the colon, and the second pair delineates the stuff after. In the substitution text, \1 and \2 are given in reverse order, to perform the swap:

```
:s/\(.*\):\(.*\)/\2:\1/
```

6.1.5. OPTIONS

Options may be set or examined via the colon command ‘‘set’’. The values of options will affect the operation of later commands.

For convenience, options have both a long descriptive name and a short name which is easy to type. You may use either name interchangeably.

long name	short name	type	default	meaning
autoindent	ai	Bool	noai	auto-indent during input
autoprint	ap	Bool	ap in *ex*,	print the current line
autowrite	aw	Bool	noaw	auto-write when switching files
charattr	ca	Bool	noca	interpret \fX sequences?
cc	cc	Str	cc="cc –c"	name of the C compiler
columns	co	Num	co=80	width of the screen
digraph	dig	Bool	nodig	recognize digraphs?
directory	dir	Str	dir="/usr/tmp"	where tmp files are kept

edcompatible	ed	Bool	noed	remember ":s//" options
errorbells	eb	Bool	eb	ring bell on error
exrefresh	er	Bool	er	write lines individually in *ex*
flipcase	fc	Str	fc=""	non-ASCII chars flipped by ~
hideformat	hf	Bool	hf	hide text formatter commands
ignorecase	ic	Bool	noic	upper/lowercase match in search
inputmode	im	Bool	noim	start vi in insert mode?
keytime	kt	Num	kt=2	timeout for mapped key entry
keywordprg	kp	Str	kp="ref"	full path name of shift–K prog
lines	ln	Num	ln=25	number of lines on the screen
list	li	Bool	noli	display lines in ‘‘list’’ mode
magic	ma	Bool	ma	use regular expression in search
make	mk	Str	mk="make"	name of the ‘‘make’’ program
modeline	ml	Bool	noml	are modelines processed?
paragraphs	pa	Str	pa="PPppIPLPQP"	names of ‘‘paragraph’’ nroff cmd
readonly	ro	Bool	noro	prevent overwriting of orig file
report	re	Num	re=5	report when 5 or more changes
scroll	sc	Num	sc=12	scroll amount for ^U and ^D
sections	se	Str	se="NHSHSSSEse"	names of ‘‘section’’ nroff cmd
shell	sh	Str	sh="/bin/sh"	full path name of the shell
showmatch	sm	Bool	nosm	show matching ()[]{}
showmode	smd	Bool	nosmd	say when in input mode
shiftwidth	sw	Num	sw=8	shift amount for < and >
sidescroll	ss	Num	ss=8	amount of sideways scrolling
sync	sy	Bool	nosy	call *sync*() often
tabstop	ts	Num	ts=8	width of tab characters
term	te	Str	te="$TERM"	name of the termcap entry
vbell	vb	Bool	vb	use visible alternative to bell
warn	wa	Bool	wa	warn for ! if file modified
wrapmargin	wm	Num	wm=0	wrap long lines in input mode
wrapscan	ws	Bool	ws	at EOF searches wrap to line 1

There are three types of options: Bool, string, and numeric. Boolean options are made TRUE by giving the name of the option as an argument to the ‘‘set’’ command; they are made FALSE by prefixing the name with ‘‘no’’. For example, ‘‘set autoindent’’ makes the autoindent option TRUE, and ‘‘set noautoindent’’ makes it FALSE.

To change the value of a string or numeric option, pass the ‘‘set’’ command the name of the option, followed by an ‘‘=’’ sign and the option’s new value. For example, ‘‘set tabstop=8’’ will give the tabstop option a value of 8. For string options, you may enclose the new value in quotes.

AutoIndent

During input mode, the autoindent option will cause each added line to begin with the same amount of leading white space as the line above it. Without autoindent, added lines are initially empty.

AutoPrint

This option only affects *ex* mode. If the autoprint option on, and either the cursor has moved to a different line or the previous command modified the file, then *elvis* will print the current line.

AutoWrite

When you are editing one file and decide to switch to another – via the :tag command, or :next command, perhaps – if your current file has been modified, then *elvis* will normally print an error message and refuse to switch.

However, if the autowrite option is on, then *elvis* will write the modified version of the current file and successfully switch to the new file.

CC

The :cc command runs the C compiler. This option should be set to the name of your compiler.

CharAttr

Many text formatting programs allow you to designate portions of your text to be underlined, italicized, or boldface by embedding the special strings \fU, \fI, and \fB in your text. The special string \fR marks the end of underlined or boldface text.

Elvis normally treats those special strings just like any other text.

However, if the charattr option is on, then *elvis* will interpret those special strings correctly, to display underlined or boldface text on the screen. (This only works, of course, if your terminal can display underlined and boldface, and if the termcap entry says how to do it.)

COlumns

This is a ‘‘read only’’ option. You cannot change its value, but you can have *elvis* print it. It shows how wide your screen is.

DIGraph

This option is used to enable/disable recognition of digraphs. The default value is nodigraph, which means that digraphs will not be recognized.

DIRectory

Elvis stores text in temporary files. This option allows you to control which directory those temporary files will appear in. The default is */usr/tmp*.

This option can only be set in a *.exrc* file; after that, *elvis* will have already started making temporary files in some other directory, so it would be too late.

EDcompatible

This option affects the behavior of the ‘‘:s/regexp/text/options’’ command. It is normally off (:se noed) which causes all of the substitution options to be off unless explicitly given.

However, with *edcompatible* on (:se ed), the substitution command remembers which options you used last time. Those same options will continue to be used until you change them. In edcompatible mode, when you explicitly give the name of a substitution option, you will toggle the state of that option.

ErrorBells

Elvis normally rings a bell when you do something wrong. This option lets you disable the bell.

ExRefresh

The *ex* mode of *elvis* writes many lines to the screen. You can make *elvis* either write each line to the screen separately, or save up many lines and write them all at once.

The exrefresh option is normally on, so each line is written to the screen separately.

You may wish to turn the exrefresh option off (:se noer) if the ‘‘write’’ system call is costly on your machine, or if you are using a windowing environment. (Windowing environments scroll text a lot faster when you write many lines at once.)

This option has no effect in visual command mode or input mode.

FlipCase

The flipcase option allows you to control how the non-ASCII characters are altered by the ‘‘~’’ command.

The string is divided into pairs of characters. When ‘‘~’’ is applied to a non-ASCII character, *elvis* looks up the character in the flipcase string to see which pair it is in, and replaces it by the other character of the pair.

HideFormat

Many text formatters require you to embed format commands in your text, on lines that start with a ‘‘.’’ character. *Elvis* normally displays these lines like any other text, but if the hideformat option is on, then format lines are displayed as blank lines.

IgnoreCase

Normally, when *elvis* searches for text, it treats uppercase letters as being different for lowercase letters.

When the ignorecase option is on, uppercase and lowercase are treated as equal.

InputMode

This option allows you to have *elvis* start up in insert mode. You can still exit insert mode at any time by hitting the ESC key, as usual. Usually, this option would be set in your *.exrc* file.

KeyTime

The arrow keys of most terminals send a multi-character sequence. It takes a measurable amount of time for these sequences to be transmitted. The keytime option allows you to control the maximum amount of time to allow for an arrow key (or other mapped key) to be received in full.

The default keytime value is 2. Because of the way UNIX time-keeping works, the actual amount of time allowed will vary slightly, but it will always be between 1 and 2 seconds.

If you set keytime to 1, then the actual amount of time allowed will be between 0 and 1 second. This will generally make the keyboard's response be a little faster (mostly for the ESC key), but on those occasions where the time allowed happens to be closer to 0 than 1 second, *elvis* may fail to allow enough time for an arrow key's sequence to be received fully.

As a special case, you can set keytime to 0 to disable this time limit stuff altogether. The big problem here is: If your arrow keys' sequences start with an ESC, then every time you hit your ESC key *elvis* will wait... and wait... to see if maybe that ESC was part of an arrow key's sequence.

NOTE: this option is a generalization of the timeout option of the real *vi*.

KeywordPrg

Elvis has a special keyword lookup feature. You move the cursor onto a word, and hit shift–K, and *elvis* uses another program to look up the word and display information about it.

This option says which program gets run.

The default value of this option is ''ref'', which is a program that looks up the definition of a function in C. It looks up the function name in a file called ''refs'' which is created by ctags.

You can substitute other programs, such as an English dictionary program or the on-line manual. *Elvis* runs the program, using the keyword as its only argument. The program should write information to *stdout*. The program's exit status should be 0, unless you want *elvis* to print ''<<< failed >>>''.

LiNes

This ''read only'' option shows how many lines you screen has.

LIst

In nolist mode (the default), *elvis* displays text in a ‘‘normal’’ manner — with tabs expanded to an appropriate number of spaces, etc.

However, sometimes it is useful to have tab characters displayed differently. In list mode, tabs are displayed as ‘‘^I’’, and a ‘‘$’’ is displayed at the end of each line.

MAgic

The search mechanism in *elvis* can accept ‘‘regular expressions’’ — strings in which certain characters have special meaning.

The magic option is normally on, which causes these characters to be treated specially.

If you turn the magic option off (:se noma), then all characters except ^ and $ are treated literally. ^ and $ retain their special meanings regardless of the setting of magic.

MaKe

The :make command runs your ‘‘make’’ program. This option defines the name of your ‘‘make’’ program.

ModeLine

Elvis supports modelines. Modelines are lines near the beginning or end of your text file which contain ‘‘ex:yowza:’’, where ‘‘yowza’’ is any *ex* command. A typical ‘‘yowza’’ would be something like ‘‘set ts=4 ca kp=spell’’.

Normally these lines are ignored, for security reasons, but if you have ‘‘set modeline’’ in your *.exrc* file then ‘‘yowza’’ is executed.

PAragraphs

The { and } commands move the cursor forward or backward in increments of one paragraph. Paragraphs may be separated by blank lines, or by a ‘‘dot’’ command of a text formatter. Different text formatters use different ‘‘dot’’ commands. This option allows you to configure *elvis* to work with your text formatter.

It is assumed that your formatter uses commands that start with a ‘‘.’’ character at the front of a line, and then have a one- or two-character command name.

The value of the paragraphs option is a string in which each pair of characters is one possible form of your text formatter’s paragraph command.

ReadOnly

Normally, *elvis* will let you write back any file to which you have write permission. If you do not have write permission, then you can only write the changed version of the file to a **different** file.

If you set the readonly option, then *elvis* will pretend you do not have write permission to **any** file you edit. It is useful when you really only mean to use *elvis* to look at a file, not to change it. This way you cannot change it accidentally.

This option is normally off, unless you use the ‘‘view’’ alias of *elvis*. *View* is like *vi* except that the readonly option is on.

REport

Commands in *elvis* may affect many lines. For commands that affect a lot of lines, *elvis* will output a message saying what was done and how many lines were affected. This option allows you to define what ‘‘a lot of lines’’ means. The default is 5, so any command which affects 5 or more lines will cause a message to be shown.

SCroll

The ^U and ^D keys normally scroll backward or forward by half a screen full, but this is adjustable. The value of this option says how many lines those keys should scroll by.

SEctions

The [[and]] commands move the cursor backward or forward in increments of 1 section. Sections may be delimited by a { character in column 1 (which is useful for C source code) or by means of a text formatter’s ‘‘dot’’ commands.

This option allows you to configure *elvis* to work with your text formatter’s ‘‘section’’ command, in exactly the same way that the paragraphs option makes it work with the formatter’s ‘‘paragraphs’’ command.

SHell

When *elvis* forks a shell (perhaps for the :! or :shell commands) this is the program that is uses as a shell. This is ‘‘/bin/sh’’ by default, unless you have set the `SHELL` (or `COMSPEC`, for MS-DOS) environment variable, it which case the default value is copied from the environment.

ShiftWidth

The < and > commands shift text left or right by some uniform number of columns. The shiftwidth option defines that ‘‘uniform number’’. The default is 8.

ShowMatch

With showmatch set, in input mode every time you hit one of)}], *elvis* will momentarily move the cursor to the matching ({[and then return to the original position. If there is no matching ({[then it will beep at you.

ShowMoDe

In visual mode, it is easy to forget whether you are in the visual command mode or input/replace mode. Normally, the showmode option is off, and you have not a clue as to which mode you are in. If you turn the showmode option on, though, a little message will appear in the lower right-hand corner of your screen, telling you which mode you are in.

SideScroll

For long lines, *elvis* scrolls sideways. (This is different from the real *vi*, which wraps a single long line onto several rows of the screen.)

To minimize the number of scrolls needed, *elvis* moves the screen sideways by several characters at a time. The value of this option says how many characters' widths to scroll at a time.

Generally, the faster your screen can be redrawn, the lower the value you will want in this option.

SYnc

If the system crashes during an edit session, then most of your work can be recovered from the temporary file that *elvis* uses to store changes. However, sometimes the OS will not copy changes to the hard disk immediately, so recovery might not be possible. The [no]sync option lets you control this.

In nosync mode (which is the default, for UNIX), *elvis* lets the operating system control when data is written to the disk. This is generally faster.

In sync mode (which is the default, for MS-DOS), *elvis* forces all changes out to disk every time you make a change. This is generally safer, but slower. It can also be a rather rude thing to do on a multi-user system.

TabStop

Tab characters are normally 8 characters wide, but you can change their widths by means of this option.

TErm

This ''read only'' option shows the name of the termcap entry that *elvis* is using for your terminal.

VBell

If your termcap entry describes a visible alternative to ringing your terminal's bell, then this option will say whether the visible version gets used or not. Normally it will be.

If your termcap does **not** include a visible bell capability, then the vbell option will be off, and you cannot turn it on.

WArn

If you have modified a file but not yet written it back to disk, then *elvis* will normally print a warning before executing a ‘‘:!cmd’’ command. However, in nowarn mode, this warning is not given.

Elvis also normally prints a message after a successful search that wrapped at EOF. The [no]warn option can also disable this warning.

WrapMargin

Normally (with wrapmargin=0) *elvis* will let you type in extremely long lines, if you wish.

However, with wrapmargin set to something other that 0 (wrapmargin=10 is nice), *elvis* will automatically cause long lines to be ‘‘wrapped’’ on a word break for lines longer than wrapmargin’s setting.

WrapScan

Normally, when you search for something, *elvis* will find it no matter where it is in the file. *Elvis* starts at the cursor position, and searches forward. If *elvis* hits EOF without finding what you are looking for, then it wraps around to continue searching from line 1.

If you turn off the wrapscan option (:se nows), then when *elvis* hits EOF during a search, it will stop and say so.

6.1.6. CUT BUFFERS

When *elvis* deletes text, it stores that text in a cut buffer. This happens in both visual mode and *ex* mode. There is no practical limit to how much text a cut buffer can hold.

There are 36 cut buffers: 26 named buffers ("a through "z), 9 anonymous buffers ("1 through "9), and 1 extra cut buffer (".).

In *ex* mode, the :move and :copy commands use a cut buffer to temporarily hold the text to be moved/copied.

Filling

In visual mode, text is copied into a cut buffer when you use the d, y, c, C, or s commands.

By default, the text goes into the "1 buffer. The text that used to be in "1 gets shifted into "2, "2 gets shifted into "3, and so on. The text that used to be in "9 is lost. This way, the last 9 things you deleted are still accessible.

You can also put the text into a named buffer — "a through "z. To do this, you should type the buffer’s name (two keystrokes: a double-quote and a lowercase letter) before the d/y/c/C/s command. When you do this, "1 through "9 are not affected by the cut.

You can append text to one of the named buffers. To do this, type the buffer’s name in uppercase (a double-quote and an uppercase letter) before the d/y/c/C/s command.

The ". buffer is special. It is not affected by the d/y/c/C/s command. Instead, it stores the

text that you typed in the last time you were in input mode. It is used to implement the ‘‘.’’ visual command, and ^A in input mode.

In *ex* mode (also known as colon mode), the :delete, :change, and :yank commands all copy text into a cut buffer. Like the visual commands, these *ex* commands normally use the "1 buffer, but you can use one of the named buffers by giving its name after the command. For example,

```
:20,30y a
```

will copy lines 20 through 30 into cut buffer "a.

You cannot directly put text into the ". buffer, or the "2 through "9 buffers.

Pasting from a Cut Buffer

There are two styles of pasting: line-mode and character-mode. If a cut buffer contains whole lines (from a command like ‘‘dd’’) then line-mode pasting is used; if it contains partial lines (from a command like ‘‘dw’’) then character-mode pasting is used. The *ex* commands always cut whole lines.

Character-mode pasting causes the text to be inserted into the line that the cursor is on.

Line-mode pasting inserts the text on a new line above or below the line that the cursor is on. It does not affect the cursor’s line at all.

In visual mode, the p and P commands insert text from a cut buffer. Uppercase P will insert it before the cursor, and lowercase p will insert it after the cursor. Normally, these commands will paste from the "1 buffer, but you can specify any other buffer to paste from. Just type its name (a double-quote and another character) before you type the P or p.

In *ex* mode, the (pu)t command pastes text after a given line. To paste from a buffer other that "1, enter its name after the command.

Macros

The contents of a named cut buffer can be executed as a series of *ex/vi* commands.

To put the instructions into the cut buffer, you must first insert them into the file, and then delete them into a named cut buffer.

To execute a cut buffer’s contents as *ex* commands, you should give the *ex* command ‘‘@’’ and the name of the buffer. For example, :@z will execute "z as a series of *ex* commands.

To execute a cut buffer’s contents as visual commands, you should give the visual command ‘‘@’’ and the letter of the buffer’s name. The visual ‘‘@’’ command is different from the *ex* ‘‘@’’ command. They interpret the cut buffer’s contents differently.

The visual @ command can be rather finicky. Each character in the buffer is interpreted as a keystroke. If you load the instructions into the cut buffer via a "zdd command, then the newline character at the end of the line will be executed just like any other character, so the cursor would be moved down 1 line. If you do not want the cursor to move down 1 line at the end of each @z command, then you should load the cut buffer by saying 0"zD instead.

Although cut buffers may hold any amount of text, *elvis* can only *execute* small buffers. For *ex* mode, the buffer is limited to about 1k bytes. For visual mode, the buffer is limited to

about 80 bytes. If a buffer is too large to execute, an error message is displayed.

You cannot nest @ commands. You cannot run @ commands from your *.exrc* file, or any other :source file either. Similarly, you cannot run a :source command from within an @ command. Hopefully, these restrictions will be lifted in a later version.

The Effect of Switching Files

When *elvis* first starts up, all cut buffers are empty. When you switch to a different file (via the :n or :e commands perhaps) the 9 anonymous cut buffers are emptied again, but the other 27 buffers retain their text.

6.1.7. DIFFERENCES BETWEEN ELVIS & BSD VI/EX

Extensions

```
:mkexrc
:mk
```

This *ex* command saves the current :set and :map configurations in the *.exrc* file in your current directory.

```
:Next
:previous
:N
:pre
```

These commands move backwards through the args list.

```
zz
```

In visual command mode, the (lowercase) ‘‘zz’’ command will center the current line on the screen, like ‘‘z=’’.

The default count value for ‘‘.’’ is the same as the previous command which ‘‘.’’ is meant to repeat. However, you can supply a new count if you wish. For example, after ‘‘3dw’’, ‘‘.’’ will delete 3 words, but ‘‘5.’’ will delete 5 words.

```
".
```

The text which was most recently input (via a ‘‘cw’’ command, or something similar) is saved in a cut buffer called ". (which is a pretty hard name to write in an English sentence).

```
K
```

In visual command mode, you can move the cursor onto a word and press shift–K to have *elvis* run a reference program to look that word up. This command alone is worth the price of admission! See the ctags and ref programs.

```
#
```

In visual command mode, you can move the cursor onto a number and then hit ## or #+ to increment that number by 1. To increment it by a larger amount, type in the increment value before hitting the initial #. The number can also be decremented or set by hitting #– or #=, respectively.

`input`

You can backspace past the beginning of the line. The arrow keys work in input mode.

If you type CTRL–A, then the text that you input last time is inserted. You will remain in input mode, so you can backspace over part of it, or add more to it. (This is sort of like CTRL–@ on the real *vi*, except that CTRL–A really works.)

CTRL–P will insert the contents of the cut buffer.

Real *vi* can only remember up to 128 characters of input, but *elvis* can remember any amount.

The ^T and ^D keys can adjust the indent of a line no matter where the cursor happens to be in that line.

You can save your file and exit *elvis* directly from input mode by hitting CTRL–Z twice.

Elvis supports digraphs as a way to enter non-ASCII characters.

```
:set inputmode
:se im
```

If you set this flag in your *.exrc* file, then *elvis* will start up in input mode instead of visual command mode.

```
:set charattr
:se ca
```

Elvis can display ‘‘backslash–f’’ style character attributes on the screen as you edit. The following example shows the recognized attributes:

normal \fBboldface\fR \fIitalics\fR \fUunderlined\fR normal

NOTE: you must compile *elvis* without the `-DNO_CHARATTR` flag for this to work.

```
:set sync
:se sy
```

After a crash, you can usually recover the altered form of the file from the temporary file that *elvis* uses. With the sync option turned on, the odds are shifted a little more in your favor because *elvis* will perform a *sync*() call after each change has been written to the temporary file.

`cursor shape`

If your terminal’s termcap entry includes the necessary capabilities, *elvis* changes the shape of the cursor to indicate which mode you are in.

```
:set hideformat
:se hf
```

This option hides format control lines. (They are displayed on the screen as blank lines.)

```
:errlist
*
elvis -m
```

Elvis is clever enough to parse the error messages emitted by many compilers. To use this feature, you should collect your compiler's error messages into a file called *errlist*; *elvis* will read this file, determine which source file caused the error messages, start editing that file, move the cursor to the line where the error was detected, and display the error message on the status line.

Omissions

The replace mode is a hack. It does not save the text that it overwrites.

Long lines are displayed differently — where the real *vi* would wrap a long line onto several rows of the screen, *elvis* simply displays part of the line, and allows you to scroll the screen sideways to see the rest of it.

The ‘‘:preserve’’ and ‘‘:recover’’ commands are missing. So is the **–r** flag. Since ‘‘:recover’’ is used so rarely it was implemented as a separate program. There's no need to load the recovery code into memory every time you edit a file.

LISP support is missing.

Due to naming conventions used for the temporary files, *elvis* can be creating no more that one new file per directory at any given time. Any number of existing files can be edited at the same time on multitasking computer systems, but only one new file can be created simultaneously per directory. To relieve this problem, you would have to edit tmp.c and elvrec.c This is expected to be done in version 1.5

Autoindent mode acts a little different from the real *vi*. It is still quite useful, but if you frequently use both *vi* and *elvis* then the differences may be annoying. Autoindent is **gradually** improving.

The visual ‘‘put’’ command cannot be repeated by hitting the . key.

6.1.8. INTERNAL

You do not need to know the material in this section to use *elvis*. You only need it if you intend to modify *elvis*.

The temporary file

The temporary file is divided into blocks of 1024 bytes each.

When *elvis* starts up, the file is copied into the temporary file. Small amounts of extra space are inserted into the temporary file to insure that no text lines cross block boundaries; this speeds up processing and simplifies storage management. The ‘‘extra space‘‘ is filled with NUL characters; the input file must not contain any NULs, to avoid confusion.

The first block of the temporary file is an array of shorts which describe the order of the blocks; that is, header[1] is the block number of the first block, and so on. This limits the

temporary file to 512 active blocks, so the largest file you can edit is about 400K bytes long!

When blocks are altered, they are rewritten to a **different** block in the file, and the in-core version of the header block is updated accordingly. The in-core header block will be copied to the temp file immediately before the next change... or, to undo this change, swap the old header (from the temp file) with the new (in-core) header.

Elvis maintains another in-core array which contains the line-number of the last line in every block. This allows you to go directly to a line, given its line number.

Implementation of Editing

There are three basic operations which affect text:

- delete text – delete(from, to)
- add text – add(at, text)
- yank text – cut(from, to)

To yank text, all text between two text positions is copied into a cut buffer. The original text is not changed. To copy the text into a cut buffer, you need only remember which physical blocks that contain the cut text, the offset into the first block of the start of the cut, the offset into the last block of the end of the cut, and what kind of cut it was. (Cuts may be either character cuts or line cuts; the kind of a cut affects the way it is later ‘‘put’’.) This is implemented in the function *cut*()

To delete text, you must modify the first and last blocks, and remove any reference to the intervening blocks in the header’s list. The text to be deleted is specified by two marks. This is implemented in the function *delete*().

To add text, you must specify the text to insert (as a null-terminated string) and the place to insert it (as a mark). The block into which the text is to be inserted may need to be split into as many as four blocks, with new intervening blocks needed as well... or it could be as simple as modifying a single block. This is implemented in the function *add*().

Other interesting functions are *paste*() (to copy text from a cut buffer into the file), *modify*() (for an efficient way to implement a combined delete/add sequence), and *input*() (to get text from the user and insert it into the file).

When text is modified, an internal file-revision counter, called ‘‘changes’’, is incremented. This counter is used to detect when certain caches are out of date. (The ‘‘changes’’ counter is also incremented when we switch to a different file, and also in one or two similar situations — all related to invalidating caches.)

Marks and the Cursor

Marks are places within the text. They are represented internally as a long variable which is split into two bit-fields: a line number and a character index. Line numbers start with 1, and character indexes start with 0.

Since line numbers start with 1, it is impossible for a set mark to have a value of 0L. 0L is therefore used to represent unset marks.

When you do the ‘‘delete text’’ change, any marks that were part of the deleted text are

unset, and any marks that were set to points after it are adjusted. Similarly, marks are adjusted after new text is inserted.

The cursor is represented as a mark.

Colon Command Interpretation

Colon commands are parsed, and the command name is looked up in an array of structures which also contain a pointer to the function that implements the command, and a description of the arguments that the command can take. If the command is recognized and its arguments are legal, then the function is called.

Each function performs its task; this may cause the cursor to be moved to a different line, or whatever.

Screen Control

The screen is updated via a package which looks like the ‘‘curses’’ library, but is not. It is actually much simpler. Most curses operations are implemented as macros which copy characters into a large I/O buffer, which is then written with a single large *write*() call as part of the *refresh*() operation.

The functions which modify text (namely *add*() and *delete*()) remember where text has been modified. They do this by calling the function *redrawrange*(). The screen redrawing function, *redraw*(), uses these clues to help it reduce the amount of text that is redrawn each time.

Portability

To improve portability, *elvis* collects as many of the system-dependent definitions as possible into the config.h file. This file begins with some preprocessor instructions which attempt to determine which compiler and operating system you have. After that, it conditionally defines some macros and constants for your system.

One of the more significant macros is *ttyread(buf,*n) This macro is used to read raw characters from the keyboard. An attempt to read may be cut short by a `SIGALRM` signal. For UNIX systems, this simply reads bytes from *stdin*. For MSDOS, TOS, and OS9, *ttyread*() is a function defined in *curses.c*. There is also a *ttywrite*() macro.

The *tread*() and *twrite*() macros are versions of *read*() and *write*() that are used for text files. On UNIX systems, these are equivalent to *read*() and *write*(). On MS-DOS, these are also equivalent to *read*() and *write*(), since DOS libraries are generally clever enough to convert newline characters automatically. For Atari TOS, though, the MWC library is too stupid to do this, so we had to do the conversion explicitly.

Other macros may substitute *index*() for *strchr*(), or *bcopy*() for *memcpy*(), or map the ‘‘void’’ data type to ‘‘int’’, or whatever.

The file *tinytcap.c* contains a set of functions that emulate the termcap library for a small set of terminal types. The terminal-specific info is hard-coded into this file. It is only used for systems that do not support real termcap. Another alternative for screen control can be seen

in the *curses.h* and *pc.c* files. Here, macros named VOIDBIOS and CHECKBIOS are used to indirectly call functions which perform low-level screen manipulation via BIOS calls.

The *stat*() function must be able to come up with UNIX-style major/minor/inode numbers that uniquely identify a file or directory.

Please try to keep you changes localized, and wrap them in #if/#endif pairs, so that *elvis* can still be compiled on other systems. And please forward updates to the author so that they can be incorporated into the latest-and-greatest version of *elvis*.

6.1.9. CFLAGS

Elvis uses many preprocessor symbols to control compilation. Some of these control the sizes of buffers and such. The −DNO_XXXX options remove small sets of related features.

Most *elvis* users will probably want to keep all features available. Minix-PC users, though, will have to sacrifice some sets because otherwise *elvis* would be too bulky to compile. The ‘‘asld’’ phase of the compiler craps out.

−DM_SYSV, −DTOS, −DCOHERENT
: These flags tell the compiler that *elvis* is being compiled for System-V UNIX, Atari TOS, or Coherent, respectively. For other systems, the config.h file can generally figure it out automatically.

−DDATE=*string*
: DATE should be defined to be a string constant. It is printed by the :version command as the compilation date of the program.

  It is only used in *cmd1.c*, and even there you may leave it undefined without causing an urp.

−DNBUFS=*numberElvis* keeps most of your text in a temporary file; only a small amount is actually stored in RAM. This flag allows you to control how much of the file can be in RAM at any time. The default is 5 blocks. (See the −DBLKSIZE flag, below.)

  More RAM allows global changes to happen a little faster. If you are just making many small changes in one section of a file, though, extra RAM will not help much.

−DBLKSIZE=*number*
: This controls the size of blocks that *elvis* uses internally. The value of BLKSIZE must be a power of two. The default value is 1024, which allows you to edit files up to almost 512K bytes long. Every time you double BLKSIZE, you quadruple the size of a text file that *elvis* can handle, but you also cause the temporary file to grow faster.

−DTMPDIR=*string*
: This sets the default value of the ‘‘directory’’ option, which specifies where the temporary files should reside. The value of TMPDIR must be a string, so be sure your value includes the quote characters on each end.

−DEXRC=*str*, −DHMEXRC=*str*, −DSYSEXRC=*str*, −DEXINIT=*str*
: This lets you control the names of the initialization files. Their values must be strings, so be careful about quoting.

  EXRC is the name of the initialization file in the current directory. Its default value is *.exrc* on Amoeba. For other systems, check the config.h file.

HMEXRC is the name of the initialization file in your home directory. By default, it is the same as EXRC. *Elvis* will automatically prepend the name of your home directory to HMEXRC at run time, so do not give a full path name.

SYSEXRC is the name of a system-wide initialization file. It has no default value; if you do not define a value for it, then the code that supports SYSEXRC just is not compiled. The value of SYSEXRC should be a full path name, in quotes.

EXINIT is the name of an environment variable that can contain initialization commands. Normally, its value is ‘‘EXINIT’’.

−DKEYWORDPRG=*string*
This flag determines the default value of the ‘‘keywordprg’’ option. Its value must be a string, so be careful about quoting. The default value of this flag is ‘‘ref’’, which is a C reference program.

−DCC_COMMAND=*string* −DMAKE_COMMAND=*string* −DERRLIST=*string*
These control the names of the C compiler, the *make* utility, and the error output file, respectively. They are only used if −DNO_ERRLIST is not given.

−DMAXMAPS=*number*
This controls the capacity of the key map table.

−DMAXRCLEN=*number*
This determines how large a *.exrc* file can be (measured in bytes). The default is 1000 bytes. If you increase this value significantly, then you may need to allocate extra memory for the stack. See the CHMEM setting in the *Makefile*.

−DSHELL=*string*
This is the default value of the ‘‘shell’’ option, and hence the default shell used from within *elvis*. This only controls the default; the value you give here may be overridden at run-time by setting an environment variable named SHELL (or COMSPEC for MS-DOS). Its value must be a string constant, so be careful about quoting.

−DTAGS=*string*
This sets the name of the ‘‘tags’’ file, which is used by the :tag command. Its value must be a string constant, so be careful about quoting.

−DCS_IBMPC
The digraph table and flipcase option will normally start out empty. However, if you add −DCS_IBMPC or −DCS_LATIN1 to your CFLAGS, then they will start out filled with values that are appropriate for the IBM PC character set or the ISO Latin−1 character set, respectively.

−DDEBUG
This adds the ‘‘:debug’’ and ‘‘:validate’’ commands, and also adds many internal consistency checks. It increases the size of the ‘‘.text’’ segment by about 6K.

−DCRUNCH
This flag removes some non-critical code, so that *elvis* is smaller. For example, it removes a short-cut from the regexp package, so that text searches are slower. Also, screen updates are not as efficient. A couple of obscure features are disabled by this, too.

−DNO_MKEXRC
This removes the ‘‘:mkexrc’’ command, so you have to create any *.exrc* files manually.

The size of the .text segment will be reduced by about 600 bytes.

−DNO_CHARATTR
Permanently disables the charattr option. This reduces the size of your ‘‘.text’’ segment by about 850 bytes.

−DNO_RECYCLE
Normally, *elvis* will recycle space (from the tmp file) which contains totally obsolete text. This flag disables this recycling. Without recycling, the ‘‘.text’’ segment is about 1K smaller than it would otherwise be, but the tmp file grows much faster. If you have a lot of free space on your hard disk, but *elvis* is too bulky to run with recycling, then try it without recycling.

When using a version of *elvis* that has been compiled with −DNO_RECYCLE, you should be careful to avoid making many small changes to a file because each individual change will cause the tmp file to grow by at least 1k. Hitting ‘‘x’’ thirty times counts as thirty changes, but typing ‘‘30x’’ counts as one change. Also, you should occasionally do a ‘‘:w’’ followed by a ‘‘:e’’ to start with a fresh tmp file.

−DNO_SENTENCE
Leaves out the ‘‘(’’ and ‘‘)’’ visual mode commands. Also, the ‘‘[[’’, ‘‘]]’’, ‘‘{’’, and ‘‘}’’ commands will not recognize *roff macros. The sections and paragraphs options go away. This saves about 650 bytes in the ‘‘.text’’ segment.

−DNO_CHARSEARCH
Leaves out the visual commands which locate a given character in the current line: ‘‘f’’, ‘‘t’’, ‘‘F’’, ‘‘T’’, ‘‘,’’ and ‘‘;’’. This saves about 900 bytes.

−DNO_EXTENSIONSLeaves out the ‘‘K’’ and ‘‘#’’ visual commands. Also, the arrow keys will no longer work in input mode. (Other extensions are either inherent in the design of *elvis*, or are controlled by more specific flags, or are too tiny to be worth removing.) This saves about 250 bytes.

−DNO_MAGIC
Permanently disables the ‘‘magic’’ option, so that most meta-characters in a regular expression are **not** recognized. This saves about 3k of space in the ‘‘.text’’ segment, because the complex regular expression code can be replaced by much simpler code.

−DNO_SHOWMODE
Permanently disables the ‘‘showmode’’ option, saving about 250 bytes.

Normally, *elvis* tries to adjust the shape of the cursor as a reminder of which mode you are in. The −DNO_CURSORSHAPE flag disables this, saving about 150 bytes.

−DNO_DIGRAPH
To allow entry of non-ASCII characters, *elvis* supports digraphs. A digraph is a single (non-ASCII) character which is entered as a combination of two other (ASCII) characters. If you do not need to input non-ASCII characters, or if your keyboard supports a better way of entering non-ASCII characters, then you can disable the digraph code and save about 450 bytes.

−DNO_ERRLIST
Elvis adds a ‘‘:errlist’’ command, which is useful to programmers. If you do not need this feature, you can disable it via the −DNO_ERRLIST flag. This will reduce the .text

segment by about 900 bytes, and the .bss segment by about 300 bytes.

−DNO_ABBR
The −DNO_ABBR flag disables the ‘‘:abbr’’ command, and reduces the size of *elvis* by about 600 bytes.

−DNO_OPTCOLSWhen *elvis* displays the current options settings via the ‘‘:set’’ command, the options are normally sorted into columns. The −DNO_OPTCOLS flag causes the options to be sorted across the rows, which is much simpler. The −DNO_OPTCOLS flag will reduce the size of your .text segment by about 500 bytes.

−DNO_MODELINE
This removes all support for modelines.

6.1.10. TERMCAP

Elvis uses fairly standard termcap fields for most things. The cursor shape names are non-standard but other than that there should be no surprises.

Required numeric fields

:co#: number of columns on the screen (characters per line)
:li#: number of lines on the screen

Required string fields

:ce=: clear to end-of-line
:cl=: home the cursor & clear the screen
:cm=: move the cursor to a given row/column
:up=: move the cursor up one line

Boolean fields

:am: auto margins – wrap when a char is written to the last column?
:pt: physical tabs?

Optional string fields

:al=: insert a blank row on the screen
:dl=: delete a row from the screen
:cd=: clear to end of display
:ei=: end insert mode
:ic=: insert a blank character
:im=: start insert mode
:dc=: delete a character
:sr=: scroll reverse (insert a row at the top of the screen)
:vb=: visible bell
:ti=: terminal initialization string, to start full-screen mode

:te=: terminal termination, to end full-screen mode
:ks=: enables the cursor keypad
:ke=: disables the cursor keypad

Optional strings received from the keyboard

:kd=: sequence sent by the <down arrow> key
:kl=: sequence sent by the <left arrow> key
:kr=: sequence sent by the <right arrow> key
:ku=: sequence sent by the <up arrow> key
:kP=: sequence sent by the <PgUp> key
:kN=: sequence sent by the <PgDn> key
:kh=: sequence sent by the <Home> key
:kH=: sequence sent by the <End> key

Originally, termcap did not have any names for the <PgUp>, <PgDn>, <Home>, and <End> keys. Although the capability names shown in the table above are the most common, they are **not** universal. SCO Xenix uses :PU=:PD=:HM=:EN=: for those keys. Also, if the four arrow keys happen to be part of a 3x3 keypad, then the five non-arrow keys may be named :K1=: through :K5=:, so an IBM PC keyboard may be described using those names instead. *Elvis* can recognize any of these names.

Optional fields that describe character attributes

:so=: :se=: start/end standout mode (We do not care about :sg#:)
:us=: :ue=: start/end underlined mode
:md=: :me=: start/end boldface mode
:as=: :ae=: start/end alternate character set (italics)
:ug#: visible gap left by :us=:ue=:md=:me=:as=:ae=:

Optional fields that affect the cursor's shape

The :cQ=: string is used by *elvis* immediately before exiting to undo the effects of the other cursor shape strings. If :cQ=: is not given, then all other cursor shape strings are ignored.

:cQ=: normal cursor
:cX=: cursor shape used for reading *ex* command — steady underline
:cV=: cursor shape used for reading *vi* commands — steady block
:cI=: cursor shape used during *vi* input mode — blinking underline
:cR=: cursor shape used during *vi* replace mode — blinking block

If the capabilities above are not given, then *elvis* will try to use the following values instead.

:ve=: normal cursor, used as :cQ=:cX=:cI=:cR=:
:vs=: gaudy cursor, used as :cV=:

6.1.11. ENVIRONMENT VARIABLES

Elvis examines several environment variables when it starts up. The values of these variables are used internally for a variety of purposes. You do not need to define all of these; on most systems, *elvis* only requires `TERM` to be defined. On MS-DOS systems, even that is optional.

TERM, TERMCAP

`TERM` tells *elvis* the name of the termcap entry to use. `TERMCAP` may contain either the entire termcap entry, or the full path name of the *termcap* file to search through.

TMP, TEMP

These only work for MS-DOS and Atari TOS. Either of these variables may be used to set the ‘‘directory’’ option, which controls where temporary files are stored. If you define them both, then `TMP` is used, and `TEMP` is ignored.

EXINIT

This variable may contain a colon-mode command, which will be executed after all of the *.exrc* files but before interactive editing begins.

SHELL, COMSPEC

You can use `COMSPEC` in MS-DOS, or `SHELL` in any other system, to specify which shell should be used for executing commands and expanding wildcards.

HOME

This variable should give the full path name of your home directory. *Elvis* needs to know the name of your home directory so it can locate the *.exrc* file there.

7 Manual Pages

Below is a collection of the manual pages for the utilities and servers in everyday use by users. If installed the manual pages should be available on-line using the command *aman*(U).

Name

aal – archiver and library maintainer

Synopsis

```
aal {adrtx}[vlc] archive [filename ...]
```

Description

Aal maintains groups of ACK-object files combined into a single archive file. An index-table is automatically maintained. (A *ranlib*-like mechanism as used under UNIX is unnecessary.) The link editor *led* only understands archives made with *aal*.

The first argument specifies the actions *aal* has to perform. It consists of one character from the set **adrtx**, optionally concatenated with one or more of **vlc**. *Archive* is the archive file. The *filenames* are constituent files in the archive file. The meanings of the characters in the first argument are:

d Delete the named files from the archive file.

a Append the named files to the archive file.

r Replace the named files in the archive file. New files are placed at the end.

t Print a table of contents of the archive file. If no *filename*s are given, all files in the archive are listed. If names are given, only those files are listed.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v Verbose. Under the verbose option, *aal* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t,** it gives a long listing of all information about the files.

l Local. Normally *aal* places its temporary files in the directory /tmp. This option causes them to be placed in the local directory.

c Create. Normally *aal* will create *archive* when it needs to. The create option suppresses the normal message that is produced when *archive* is created.

Warnings

If the same file is mentioned twice in an argument list, it will be put in the archive twice.

Files

/tmp/ar.* – temporary files

See Also

The Link Editor in the Programming Languages section of the Programming Guide.

Name

ack – Amsterdam Compiler Kit

Synopsis

```
ack arguments
basic arguments
cc arguments
f77 arguments
m2 arguments
pc arguments
machine arguments
```

Description

ACK is a single compiler capable of compiling several different languages. It is also able to produce binaries for various machine architectures. The transformation of source to binary takes place in several distinct phases. Command line arguments may be used to stop the translation at any of the phases. Combining sources from several languages is allowed. The run-time system of the first language mentioned, either in the program call name (e.g., pascal) or in the arguments, is automatically included. The libraries of all other languages mentioned, containing most of the run-time systems, are also automatically included. Which combinations of languages and machines are allowed varies depending on the installation.

The actions of *ack* are to repeatedly transform files with a particular suffix into files with another suffix, finally combining the results into a single file.

Different systems may use different suffices, but the following are recognized by Amoeba:

.b Basic program.

.c C module.

.e EM assembly module in human readable form.

.k Compact EM assembly code.

.m Optimized compact EM assembly code.

.mod Modula-2 module.

.o Object file.

.p Pascal program.

.s Machine assembly language code.

Options

Ack accepts the following flags:

Output Files

–o Use the next argument as the name of the resulting file. *Ack* produces *a.out* by default. This flag can always be used when *ack* produces a single output file, as in

```
ack -c.s main.c -o new.s
```

The output is produced in the file *new.s* instead of *main.s*.

–c

–c.*suffix*

Ack tries to transform each source into a file with the given *suffix*. When no *suffix* is specified *ack* stops just before the phase where it combines all arguments into a load file, thereby transforming the sources into *.k*, *.s*, *.o* or *.m* files. One extra *suffix* is recognized here, *.i*. This tells *ack* to only preprocess all human readable sources, producing files with *suffix .i*.
Note: *ack* refuses to overwrite argument *.e* files.

–t Preserve all intermediate files. If two **–t** are given, *ack* also preserves core dumps and the output of failed transformations.

Machine Used

–m*machine*

This flag tells *ack* to generate a load file for *machine*. To generate an Amoeba binary, the argument should be ‘‘am_*arch*’’, where *arch* is the target architecture. The following command will generate an Amoeba object file *file.o* for the i80386 architecture:

```
ack -c -mam_i80386 file.c
```

–fp Use the software floating-point package.

Messages

–w Suppress all warning messages.

–E Produce a complete listing of each Pascal source program. Normally for each error, one message, including the source line number, is given.

–e List only the erroneous lines of each Pascal source program.

–v[*num*]

Verbose. Print information about the various compilation phases as they occur. The optional argument *num* specifies the amount of information printed. The higher *num*, the more information. For example, **–v3** specifies that all front and back end invocations are to be displayed.

Preprocessing

–I*dir*

‘‘#include’’ files whose names do not begin with ‘‘/’’ are always sought first in the directory of the *file* argument, then in the directories named in **–I** options, then in directories on a standard list.

–D*name=def*

–D*name*

Define the *name* to the preprocessor, as if by ‘‘#define’’. If no definition is given the *name* is defined as 1.

–U*name*
Remove any initial definition of *name*, before preprocessing.

–P Do not generate line directives.

–C Leave C-comments in.

Profiling/Debugging

–p This flag tells both the Pascal and C front ends to include code enabling the user to do some monitoring/debugging. Each time a routine is entered the routine *procentry()* is called and just before each return *procexit()* is called. These routines are supplied with one parameter, a pointer to a string containing the name of the routine. In *libamoeba.a* for Amoeba these routines generate profiling information. See *profiling*(L) for more information.

Optimizing

–O

–O*num*
Try to use the optimizers with optimization level <= *num* (default 1). Currently, only the global optimizer has a level > 1. Higher levels will invoke more passes of the global optimizer. The passes which the global optimizer may perform are the following: Inline substitution, Common subexpression elimination, Strength reduction, Use definition analysis, Live variable analysis, Register allocation, Stack pollution, Branch optimization and Cross jumping.

Also, the following flags may be used:

–s*num*
Give an indication to the inline substitution phase, how much bigger the program may get, in percentage. This is only used as a rough indication. The inline substitution phase will not make the program bigger when given **–s0**.

–a Indicate to the inline substitution phase that it is offered the whole program. This allows it to throw away routines that it has substituted inline.

–Q Give some statistics.

–T Optimize for time.

–S Optimize for size.

In principle, the optimization phases can be run in any order; a phase may even be run more than once. However, the following rules must be obeyed:

- The Live Variable analysis phase (LV) must be run prior to the Register Allocation phase (SA), as SA uses information produced by LV.
- SA should be the last phase.

Also, the following may be of use:

- Inline Substitution (IL) may create new opportunities for most other phases, so it

should be run as early as possible.

- Use Definition analysis (UD) may introduce opportunities for LV.
- Strength Reduction (SR) may create opportunities for UD.

The global optimizer is a combiner, so, when using it, offer it all the source files of the program being compiled. This is not strictly necessary, but it makes the global optimizer more effective. The current default optimization phases are:

- For –O2: CJ, BO, SP;
- For –O3: CS, SR, CJ, BO, SP, UD, LV, RA;
- For –O4: IL, CS, SR, CJ, BO, SP, UD, LV, RA;
- For higher levels: as for –O4.

–L Disable the generation of code by the front ends to record line number and source file name at run-time.

Libraries

–l*name*

Tells *ack* to insert the library module specified by *name* at this point.

–.*suffix*

When linking multiple *.o* or *.m* files created by separate calls of *ack* together, *ack* cannot deduce the run-time system needed, unless called as *pc*, *basic*, *m2* or *cc*. This flag serves to tell *ack* which run-time system is needed in such a case. For example:

```
ack -c x.c ; ack -.c x.o
```

–r.*suffix*

Most front ends and back ends use one or more run-time libraries. These flags tell *ack* to include the libraries needed when a file with *suffix* is be included in the arguments.

–LIB

This flag tells the peephole optimizer *em_opt* to add information about the visibility of the names used in each output module. This is needed by assembler/linkers when these modules are to be inserted in libraries.

General

–R*program=xxx*

Replace the *program* by the path name *xxx*. The program names referred to later in this manual are allowed here.

–R*program–xxx*

The flag argument *–xxx* is given to *program*. For example,

```
ack -.c -Rcv-s32 main.o
```

will cause the binary conversion program *cv* to be called with the option **–s32.** As a result, the main thread of the program will be provided with a 32 K stack (the default is 8 K).

–R*program:n*

Set the priority of the indicated transformation to *n*. The default priority is 0, setting it to -1 makes it highly unlikely the phase will be used, setting it to 1 makes it very likely that the phase will be used.

–k Do not stop when an error occurs, but try to transform all other arguments as far as possible.

All arguments without a suffix or with an unrecognized suffix are passed to the loader. Unrecognized flags are also passed to the loader.

Preprocessor

All C source programs are run through the preprocessor before they are fed to the compiler proper. On larger machines, the compiler has a built-in preprocessor. Other human readable sources (Modula-2, Pascal, or Basic programs and machine assembly) are only preprocessed when they start with a ‘‘#’’.

Ack adds a few macro definitions when it calls the preprocessor. These macros contain the word and pointer size and the sizes of some basic types used by the Pascal, Basic and/or C compiler. All sizes are in bytes.

Macro	Description
_EM_WSIZE	word size
_EM_SSIZE	size of shorts (C)
_EM_FSIZE	size of floats (C)
_EM_PSIZE	pointer size
_EM_LSIZE	size of longs (C+Pascal)
_EM_DSIZE	size of doubles (C+Pascal)

The name of the *machine* architecture or something like it when the machine name is numeric is also defined (as 1).

The default directories searched for include files differ for each operating system.

Programs

Ack uses one or more programs in each phase of the transformation. The table below gives the names *ack* uses for these programs. Internally *ack* maintains a mapping of these names to path names for load files. The table specifies which type of files are accepted by each program as input and the file type produced as output.

Input	Name	Output	Description
.c	cem	.k	C front end
.p	pc	.k	Pascal front end
.b	abc	.k	Basic front end
.mod	m2	.k	Modula-2 front end
.e	encode	.k	Compactify EM assembly code
.k	opt	.m	EM peephole optimizer
.k .m	decode	.e	Produce human readable EM code
.m	ego	.gk	EM global optimizer
.gk	opt2	.g	Second EM peephole optimizer
.m .g	be	.s	Back end
.s	asopt	.so	Target optimizer
.s.so	as	.o	Assembler, relocatable object
.o	led	.out	Linker, Ack object format
.out	cv	a.out	Conversion from Ack to machine binary

Files

/profile/module/ack/lib/descr/fe – front end description table

/profile/module/ack/lib/`mach`*/descr:* back end description table for *mach*.

Environment Variables

ACKDIR If set, this environment variable overrides *ack*'s idea of its home directory.

ACKM If set, this environment variable overrides *ack*'s idea of the default machine it compiles for.

ACKFE If set, this environment variable tells *ack* where to get the front-end description file.

Diagnostics

The diagnostics are intended to be self–explanatory.

Warnings

When using separate compilation, be sure to give *ack* consistent **–m** options. The linker is not able to check that all object files are indeed of the same architecture.
Not all warning messages are switched off by the **–w** flag.
Argument assembly files are not preprocessed when fed into the universal assembler/loader.

See Also

basic(U), cc(U), f77(U), m2(U), pc(U).
In the Programming Guide see the reference information on ACK in the *tools* section.

Name

ail – Amoeba Interface Language

Synopsis

```
ail [ –?bBcCdgGn ] [ –D name[=val] ] [ –f number ] [ –i inlist ]
    [ –I dir ] [-l language] [ –o dir ] [ –p preprocessor ]
    [ –U name ] [ –v version ] [ + code ]...  [ Source ]
    [ + code ]...
```

Description

Ail is the Amoeba stub-compiler. It is used to generate RPC stubs for both clients and servers. For most simple types it automatically deals with marshaling and unmarshaling of data. It reads its input through the C-preprocessor.

Options

–? Print a usage string.

–b Bypass cpp. Useful to avoid *popen()* or replace */lib/cpp*.

–B Rename all files that otherwise would be overwritten as *old-filename*.BAK. Any existing .BAK files that get in the way will be deleted.

–C Stop the macro preprocessor from clobbering comments.

–d This option sets the debug-flag. With this option *ail* reports debugging information on *stdout*.

–D name[=value]
Define the name to the preprocessor, as if by ‘‘#define’’.

–f number
Make sure no output file name is longer than *number*. This is done by deleting the characters just before the first dot.

–g Display a list of generators.

–G Like **–g**, but lists the generators flagged with an underscore. To list all the generators, use **–gG**.

–i inlist
Save the names of all files that were read to the file *inlist*. This is done by parsing the #line directives in the output of cpp. Each file name is listed once.

–I dir
Put dir in the path that cpp searches to find #include files.

–l language
Set the default output language.

–n Do not generate output.

–o dir
This flag tells where the output should go by default. If the named directory does not exist, *ail* will create it. Note that if the source uses the ‘‘Output_directory’’, the default

is not used.

−p preprocessor
: Use a different preprocessor instead of */lib/cpp*.

−U name
: Remove any initial definition of the preprocessor symbol *name*.

−v version
: Specifies the version of the AIL to C mapping. Legal versions are currently 3.5 and 4.0. The default is 4.0. From version 4.0 onward, the transaction buffer for a generated server main loop is allocated with *malloc* rather than declared on the stack. Also, starting with version 4.0, array upper bounds are passed to a server's implementation functions. This is not the case for version 3.5.

+ code
: Use *code* as input. This input is not preprocessed.

Warnings

Ail is currently unable to generate code to marshal unions, pointer types, function types, floating point types or void. Enumerated constants must be declared in increasing order, that is, enum { two=2, one=1 } is illegal.

See Also

rpc(L),
In the Programming Guide see the reference information on AIL in the *tools* section.

Name

ainstall – transform an Amoeba binary on UNIX into an Amoeba executable file

Synopsis

```
ainstall [–b extra-bss] [–f filesvr] [–s stacksize]
[–T address] unix-file [amoeba-file]
```

Description

Ainstall is *only* run under UNIX. It creates an Amoeba executable file from loader output produced under UNIX. (It is not needed when running under native Amoeba.) The output file will reside on an Amoeba file server; typically, this is the Bullet Server.

The following input formats are supported:

- a.out binaries created with the native UNIX compiler and loader
- a.out binaries created with the GNU compiler and loader.
- Amoeba binaries created with ACK. These require no further conversion.

When conversion is required, the output is produced by concatenating an Amoeba-specific data structure — the *process descriptor* — with the original contents of the UNIX executable file. Offsets for the text and data segments and the symbol table segment in the process descriptor give the offsets relative to the beginning of the file.

Options

–b bss-extra
: Specifies the number of kilobytes of extra bss (uninitialized data, preset to zero) allocated for the executing program. This is in addition to the bss needed for uninitialized global and static variables declared in the program; the extra space is used by *malloc*. Normally, the bss segment is automatically grown (or a new segment allocated) when *malloc* needs more memory, so it is not necessary to specify this option even for programs with large dynamic memory requirements. The default is 16.

–f filesvr
: Specifies the file server where the output file is created. The default is the standard Bullet Server, specified by DEF_BULLETSVR in *ampolicy.h* (typically */profile/cap/bulletsvr/default*).

–s stacksize
: Specifies the stack size (in kilobytes) for the executing program. Since Amoeba (currently) does not grow the stack automatically, this must be specified for programs that engage in deep recursion or declare large buffers on the stack. The default is zero; in this case a default is chosen by *exec_file*(L) when the program is executed (normally 16 KB or the page size, whichever is the larger).

–T address
: Specifies that the text segment must be loaded at the given address (in hexadecimal). If

the loader was given a load address then the **–T** option must be specified, and with the same address, since this information is not contained in the *a.out* header. The default is architecture-dependent, and matches the loader default, except that on a VAX the default is 400 (hex).

unix-file
: Specifies the input file. This must be in one of the formats documented in *a.out.h*(5), or an Amoeba-format binary. There is no default.

amoeba-file
: Optionally specifies the name of the output file (this is unrelated to the file server used). The default is the same as *unix-file*.

Example

```
ainstall a.out /home/hello
```

See Also

ax(U), exec_file(L), exec_findhost(L), malloc(L), proc(L).

UNIX man pages: ld(1), a.out(5).

Name

am_pd – determine if a file contains an Amoeba process descriptor

Synopsis

```
am_pd file [arch]
```

Description

Am_pd is used to determine if the *file* specified contains an Amoeba process descriptor. This will be the case if the file contains an executable binary or a core dump.

The program normally produces no output but simply returns exit status 0 if the file contains a process descriptor and 1 otherwise. If the optional *arch* argument is given then an exit status of 0 will only be returned if the process descriptor is for the specified architecture.

Am_pd is typically used in shell scripts.

Diagnostics

If *file* cannot be opened or read then a message will be printed on *stderr* to indicate why not.

Example

```
file=/bin/cat/pd.i80386
if am_pd $file i80386
then
    echo "$file is a 386 binary"
else
    echo "$file is not a 386 binary"
fi
```

Name

amake – (Amoeba make) stand-alone software configuration manager

```
amake [-f file] [-L dir] [-D var[=value]] [-v[level]] [-t file]
      [-p par] [-r] [-c] [-C] [-k] [-n] [-s] [-w] [cluster] ...
```

Description

Amake is a software configuration manager that was designed to be a tool in the same spirit as *make*, i.e., it is a stand-alone tool that, when run by the user, should invoke precisely those commands that (re-)create a set of target files, according to a description file. The main idea is to make use of previous results as much as possible. Unlike standard *make*, *amake* is also able to exploit possible parallelism between the commands.

In contrast to a lot of ‘‘extended makes,’’ the specification language is not a superset of *make*’s, but a totally new one. For a complete description of the specification language refer to the *Amake User Reference Manual* in the Programming Guide. Here we only will give a short overview, illustrated with an example at the end.

In *amake*, a software configuration can be specified by means of one or more *cluster* definitions. A cluster specifies the *targets* to be constructed from a given set of *sources.* It is *amake*’s task to deduce which *tools* to use, in what order, and how to do it efficiently. The tools available can be defined by the user, but generally a reference to a standard tool library – consisting of a set of *amake* description files – will suffice. In order to be able to decide whether a given file can be used or produced by a certain tool, files are represented as *objects*, each having a set of *attributes* providing pieces of information about the object.

A tool definition, which resembles a function definition found in common programming languages, contains in its header references to the expected *values* of certain attributes, such as ‘type’. Rather than letting the user specify the values of all the necessary attributes, these values can usually be derived by *amake* itself, using *attribute derivations*. Rules declaring the possible derivations will generally be present in the library, together with the tools that refer to the attribute values in question, but the user can provide additional rules.

When invoked, *amake* will bring the specified set of clusters up to date by applying only those tools whose inputs have changed since the previous *amake* invocation (if there was any). If an argument (e.g., a compilation flag) is changed between two invocations, *amake* will notice this – because it keeps track of previously executed tools – and will thus re-run the tool with the new arguments. Moreover, old intermediate files produced are retained by default, so changing flags back again will cause it (when all inputs are unchanged) to reinstall the previous outputs, rather than to invoke the tool again. These files are kept in a subdirectory (the *object pool*) residing in the directory from which *amake* is invoked. This subdirectory also contains the *statefile* describing previous tool-invocations.

For a full description of the *amake* language, and *amake* usage in general, refer to the *Amake User Reference Manual*. Another good starting point, if interested in the various constructions available, would be to have a look at the standard tool library.

Options

The options available are the following:

–f *file*

The default specification file *amake* reads is *Amakefile* in the context in which it is invoked. A specification file with a different name can be specified by this option. Only one **–f** option may be specified.

–L *dir*

Files having relative path names, that are to be %included are normally looked for in the default system directory (see the Files section) and the current working directory. Invoking *amake* with **–L** options causes it to look in the specified directories first, in the order in which the options are supplied.

–p *par*

By default *amake* runs at most 4 commands in parallel, to prevent overloading the local machine from thrashing. A different limit (in the range 1 to 20) can be specified with this option. When the local machine has little memory, a lower number will probably give better performance, while a higher number will be appropriate when several remote machines or (under Amoeba) processors are available. Supplying 0 as argument to this option causes *make*-like behavior; only one command is started, and *amake* waits for that command before it continues. When **–p 1** is given, one command is run in the *background*, and *amake* does some useful work in the meantime.

–r

Under UNIX, this option causes remote execution on a pool of workstations. A default pool is compiled into *amake*, but a different one can be specified in the environment variable `RSH_POOL`. Note that when using this option, be sure that all the machines in the pool are up, and also (with NFS) that your *$HOME/.rhosts* contains the machine on which *amake* is running. Furthermore, each machine has got to have the same file structure, i.e., if on the machine running *amake* a file is reachable under the name */usr/users/versto/src/monkee.c*, the same should be the case for each of the other machines in the pool.

–D*var*[=*expression*]

An *amake* variable can be set on the command line by means of a **–D** option. The *expression* is optional (default %true), and must be specified in *amake* syntax. Take care to quote the expression properly, if it contains characters with a special interpretation in the shell. Also, the argument required by the **–D** option has to be one string, so

```
amake -D src = hello.c
```

will not be interpreted in the desired way. In the Bourne shell, specify

```
amake -D 'src = hello.c'
```

(i.e., explicitly made one string) or

```
amake -Dsrc=hello.c
```

(i.e., without spaces).

–k

By default, intermediate files generated while creating the targets specified, are kept ‘‘hidden’’ in the *.Amake* directory, and moved to and from this object pool whenever necessary. The **–k** option leaves intermediate files in the current working directory. This may be used, for instance, when they contain information the user needs for debugging purposes.

–n

Stops *amake* from actually executing commands for targets that need updating. The commands are only written on standard output. It is advisable to use this option (possibly in combination with the **–v** flag, see below) when a new *Amakefile* is used for a certain configuration.

–v[*level*]

Causes *amake* to be more verbose than usual. This option is helpful for finding out why *amake* does not do what the user intends. Although *amake* is relatively intelligent, it is not clairvoyant. The desired verbosity can be specified as parameter to this option: the higher the number, the more verbosity. The default verbosity level is 1. In this case, *amake* reports, among other things, the reason why it considers a tool not to be applicable within some cluster.

–t *file*

Causes *amake* to consider files having the same last component as the argument of this option to be updated (or ‘‘touched,’’ in UNIX terms). This can be used when a command executed in a previous run did not deliver a non-zero exit status, even though it failed. Tools having a touched input will be re-invoked by *amake*. For example,

```
amake -t source.c
```

will force a recompilation of files called source.c, while

```
amake -t source.o
```

will only force re-invocation of the loader.

–w

Suppress warnings about questionable constructions, unsuspected situations, etc.

–s

By default *amake* prints the commands it is executing on standard output. Specifying this option suppresses this.

–c

This option should be specified if *amake*’s feature of caching objects stemming from previous tool invocations with different options should not be used. Generated files not needed in the current run will be removed from the cache. This option is handy when short on disk space. It also functions as a kind of ‘garbage collection’, e.g., when a target has been created with some exotic option that will not be needed again in the near future. Ideally, *amake* ought to have some clever aging algorithm to do this automatically (and it probably will, one day.)

–C

This a more drastic version of the **–c** option; it removes the entire *amake* object pool, statefile and targets. This can be used when a software configuration is ready, and its targets installed. When an *amake* bug or some external cause has damaged the statefile

containing the *amake* administration, this will not work. In this case use the command:

```
rm -rf .Amake target ...
```

cluster...

All trailing arguments are taken as names of clusters or targets that are supposed to be updated. By default the first cluster encountered is assumed to require updating, together with the clusters producing sources for it.

Diagnostics

Commands are reported on standard output as they are executed, prefixed by the machine where the execution takes place. When all commands are executed locally, they are prefixed with a virtual processor name. The diagnostics of a command are buffered, i.e., redirected to temporary files and shown when the command has finished.

Other diagnostics, warnings and error messages produced by *amake* itself, are intended to be self-explanatory. Not all of them are but they will be described once things have settled down.

Environment Variables

Under UNIX, the environment variable `RSH_POOL` can be used to specify the machines available for remote execution of commands (see the **–r** option). The names of the machines should be separated by white space.

The *amake* language contains %import directives, which can be used to set *amake* variables according to the environment *amake* is started from. Likewise, %export directives are available to put *amake* variables in the environment supplied to commands invoked by *amake*.

Files

Amakefile	Default name of the configuration description file.
.Amake	A subdirectory in the working directory from which *amake* is called. It contains files (possibly several versions of some) resulting from previous tool invocations.
.Amake/Statefile	Contains a description of invocations made in previous builds.
/usr/lib/amake	On Amoeba, this directory contains a set of files defining tools used for software development in C, including the parser generators *yacc* and *LLgen*. All that should be necessary is to put `%include std-amake.amk` at the top of the *Amakefile*.
/usr/local/lib/amake	Under UNIX, this is the default directory where *amake* looks for *amake* definition files.

Warnings

As *amake* decides where and how it starts a command, it is capable of deciding whether an argument needs quotation. (This may be the case when a shell is used.) If, for example, the C-compiler is to be called with an option defining a string, it should be specified with

```
CFLAGS = '-DSTR="string"';
```

When, as is currently the case under UNIX, the Bourne Shell is used to start the C-compiler, *amake* will insert backslashes before the " characters in the command.

The fact that there is only one object pool per context allows previous invocations to be shared between different *Amakefile*s. It is not advisable, however, to do this for specification files that have no sources in common. In that case a large statefile has to be read (and possibly written) at each *amake* invocation, resulting in poorer performance.

Amake is known to use quite a lot of memory when updating a large configuration, or when it has to maintain many instances of the same configuration. In the latter case, it helps to invoke *amake* with the **–c** flag (described above) once in a while.

Targets created with *amake* are hard-linked into the cache. Therefore be sure to copy (rather than move) a target when installing it. Otherwise a future *amake* invocation may cause an inconsistent version to overwrite the installed one.

The remote execution option should only be used to update a working configuration, because it was (only recently) discovered that the UNIX *rsh* command fails to deliver a bad exit status when a remote command fails.

Example

The following shows what the specification of simple software configuration might look like in *amake*.

```
%include std-amake.amk;

%cluster
{
   %targets comp[type = program];
   %sources parse.y, scan.l, comp.c, defs.h, comp.a;
   %use     cc-c(flags => '-DAMOEBA');
};
```

The %include directive on the first line causes *amake* to read the file *std–amake.amk*, which in turn lets it %include a standard set of tool definitions and derivation rules. The type-attribute of the target ‘‘comp’’, defined on line 5, is added explicitly and forces *amake* to apply a loader as last step in the construction of targets. The sources, introduced on line 6, have type *yacc-src*, *lex-src*, *C-src*, *C-incl* and *library* respectively, all of which can be derived using analysis of the suffices. Line 7 shows that we can influence tool-behavior (in this case of the C-compiler) by supplying other values for parameters than the default ones. Alternatively, an assignment to the global variable CFLAGS could be made, which (like in *make*) is used to alter the default argument globally.

See Also

In the Programming Guide
Amake User Reference Manual.

Name

aman – display manual pages on screen

Synopsis

```
aman [-A|H|L|T|U|k] keyword
```

Description

Aman is used to find information from the on-line manual pages. Without options it looks for a manual page with the name *keyword*. If none exists then it defaults to the **–k** option (see below) and prints a list of manual pages that refer to the specified *keyword*. If one such manual page exists it is displayed using the default pager (see *Environment Variables* below). If more than one manual page exists with that name, for example *soap*(A) and *soap*(L), then it will complain about the ambiguity, and give the various possibilities. In this case the desired manual page can be obtained using the flag corresponding to the section of the manual desired. In the above example, if the system administration manual page for the Soap Server is desired then the command

```
aman -A soap
```

will cause it to be displayed.

Options

At most one option can be given. If no manual page entry is found for the specified *keyword* then *aman* defaults to the **–k** option and any other option will be ignored.

–A This requests that only manual pages from the System Administration Guide be displayed.

–H This requests that only manual pages about include files be displayed.

–L This requests that only manual pages from the Programming Guide be displayed.

–T This requests that only manual pages about test programs be displayed.

–U This requests that only manual pages from the User Guide be displayed.

–k This requests that a list of manual pages which reference the specified *keyword* be displayed, rather than an individual manual page.

Environment Variables

The string environment variable PAGER specifies which pager will be used when displaying the manual page.

Files

`amdir`/lib/man/filenames and *`amdir`/lib/man/keywords* contain the database used to search on keywords and filenames.

The directories *`amdir`/lib/man/*/manpages* contain the *cat*able versions of the manual pages. Under Amoeba only these versions can be used.

Administration

The program *`amdir`/lib/man/catman* should be run to bring the keywords database up to date if local or new manual pages are added or modified. The *catman* program takes no arguments and runs only under UNIX. (This is because it uses *nroff* which is not available under Amoeba.) The new manual pages can be installed under Amoeba using *amdumptree*(A). The best solution is probably to set up *cron* (see *cronsubmit*(U)) to run *catman* once per week. It takes a little over five minutes to regenerate the database. Generating the database the first time is quite slow if the *cat*able manual pages must be generated. It can take up to twenty-five minutes.

Examples

```
aman boot
```

will print the manual page for the boot server.

```
aman -k boot
```

will print a list of manual page names and their synopsis of all the manual pages that have something to do with the keyword *boot*. For example, it may print the following:

```
boot(A)       - keep system services available
bootutil(A)   - boot-server control
iboot(A)      - inspect or  install  the  boot  server's  virtual
installboot(A)- install hard disk or floppy bootstrap  loader  on
isaboot(A)    - boot mechanisms for i80386 machines with  an  ISA
isaprom(A)    - convert an 80386 kernel image to ISA boot ROM im-
mkbootpartn(A)- make a bootable floppy or hard disk partition
mkkdir(A)     - make kernel bootstrap directory on disk
prkdir(A)     - print the contents of a kernel boot directory
rarp(A)       - reverse Address Resolution Protocol daemon
reboot(A)     - reboot a machine with a new kernel binary.
tftp(A)       - the Trivial File Transfer Protocol server
```

Note that the summary strings are truncated if necessary to ensure that they fit on one line.

See Also

amdir(U), cronsubmit(U).

Name

amcc – compile simple Amoeba programs under UNIX

Synopsis

```
amcc [amccopts] [ccopts] [ldopts] [ofiles] [lfiles] cfiles
```

Description

Amcc is a command which only runs under UNIX. It is a special front end to various C compilers. It is useful for compiling simple C programs, for prototyping and for compiling test programs, where the effort of generating an *Amakefile* can usefully be avoided. Programs can be compiled to run under UNIX (assuming they are not multithreaded) or they can be compiled to run under Amoeba. The default is for Amoeba. In this case it is necessary to copy the compiled program to the Amoeba file server before it can be run. See *ainstall*(U) and *tob*(U) for details about how to do this. When a program is compiled for Amoeba the option `-DAMOEBA` is passed to the C compiler. File names which end in *.o* (*ofiles*) or *.a* (*lfiles*) are passed directly to the loader. File names which end in *c* (*cfiles*) are first given to *cc*.

Options

The options are divided into several classes. The first are those which are used directly by *amcc* (*amccopts*). Then there are standard arguments for the C compiler *cc*(U) (*ccopts*). The remaining arguments are those which are passed to the loader (*ldopts*).

Options used by *amcc*.

–ack
: This causes the Amsterdam Compiler Kit (ACK) compiler (see *ack*(U) and *cc*(U)) to be used instead of the local operating system's default *cc* program. To use this it is necessary to have the UNIX version of ACK installed.

–ajax
: This option causes the *Ajax* POSIX emulation library to be linked with the binary. This option cannot be used in conjunction with the **–unix** option below.

–noexec
: This causes *amcc* to print out what it would do if it ran without this flag. It does not actually execute the commands it displays.

–i80386
: This makes the target architecture for the compilation the Intel 80386/80486 architecture. It also implies **–ack**.

–x11
: This causes the X windows library to be linked with the program.

–conf *path*
: Normally the libraries are extracted from *`amdir`/conf/amoeba* but an alternative

configuration tree can be specified by *path*.

−src *path*
The include files for the compilation are taken per default from *`amdir`/src/h*. An alternative source tree (and thus include files) is specified by *path*.

−unix
The target system for the compilation is not Amoeba but the current UNIX system where *amcc* is running.

−v
Verbose mode. *Amcc* gives a running commentary about each phase of the compilation.

Options passed directly to *cc*.

−c

−C

−D*define*

−I*include-path*

−U*define*

−E

−S

−g

−O

−p

−pg
All these options have the same meaning independent of the C compiler used; either ACK or the local UNIX C compiler.

Options passed to the loader.

−y*symbol*
This causes the loader to print the names of the object files where *symbol* is referenced.

−e *entry-point*
This sets the entry point for the program to *entry-point*. This is not advisable since the program probably will not run.

−o *file-name*
The *file-name* argument specifies the name of the binary file to be produced. If no **−o** option is present the name of the binary produced will be *a.out*.

−l*library-name*
This causes the library *library-name* to be linked with the program.

Diagnostics

If the UNIX system has no command to specify its architecture this command will complain about its absence. Missing arguments such as the path name to the **−conf** will generate appropriate error messages. For the rest the error messages are those of *cc* and the loader.

Files

The include files, unless otherwise specified, are taken from *`amdir`/src/h*. The Amoeba libraries, unless otherwise specified, are taken from *`amdir`/conf/amoeba* for Amoeba programs and from *`amdir`/conf/unix* for UNIX programs.

Warnings

Trying to compile programs to run under UNIX that have no real chance of working there, such as multithreaded programs, will almost certainly fail. If by chance they do compile, they will not work.

Examples

If the files *f1.c* and *f2.c* contain C sources for a program that is to run under Amoeba, uses the *Ajax* POSIX emulation and they need to be compiled using the ACK compiler for the 386/486 architecture then the following command will compile them.

```
amcc -i80386 -ajax -o ftest f1.c f2.c
```

The resultant binary will be in the UNIX file *ftest*. This can be installed under Amoeba using *ainstall*(U).

If the program in *f3.c* is to be compiled to run under UNIX using the local *cc* then it could be compiled as follows.

```
amcc -unix -o f3test f3.c
```

This will produce a UNIX executable called *f3test*.

See Also

ainstall(U), amake(U), amdir(U), cc(U), tob(U).

Name

amdir – print root of Amoeba distribution tree

Synopsis

```
amdir
```

Description

Amdir prints the path name of the root of the Amoeba distribution tree for the system upon which it is installed. The command is typically used by shell scripts that must examine sources, for example, *amwhereis*(U). It is also useful with commands such as *amoebatree*(A).

Example

The command

```
amdir
```

under UNIX might print

```
/user/amoeba
```

and under Amoeba it typically prints

```
/profile/module/amoeba
```

Name

amsh – start an Amoeba shell running in a UNIX tty.

Synopsis

```
amsh [-m host] [-E stringenv] [-C capenv] [-t tty] [-s shell]
                  [-d] [cmd [args]]
```

Description

Amsh is used under UNIX to start a shell running under Amoeba which uses a UNIX tty for its input and output. Typically *amsh* is started in a window. It uses the FLIP driver in the UNIX kernel to communicate with the Amoeba processor pool. It sets up a sensible capability and string environment for the shell, using only what is necessary from the UNIX string environment. It starts a session server if none is running (see *session*(U)) and then the specified shell is started. The output from the session server is written to the console if it is writable, and otherwise to the UNIX file *$HOME/.session.out*.

If a command is specified as the last argument then it will be started instead of just an interactive shell. The command may have arguments.

The Amoeba root directory (i.e., /) used is specified by the capability in the UNIX file *$HOME/.capability*.

Options

The **–m**, **–C** and **–E** options are passed straight to *ax*(U) which is used to start the shell. See *ax*(U) for more details of the exact semantics of these options.

–C *capenv*
: This allows values to be set in the capability environment of the shell to be started. The *capenv* argument is normally of the form *name=capability* but see *ax*(U) for more details.

–E *stringenv*
: This is used to set values in the string environment of the shell to be started. The *stringenv* argument is normally of the form *name=string*. See *ax*(U) for other possibilities.

–d Set debugging on. This causes *amsh* and *ax* to be verbose. It is not recommended.

–m *host*
: This option causes *amsh* to start the session server and shell on the specified *host*. The default is to let the *run*(A) server decide where to run the shell.

–s *shell*
: This causes the shell specified by the *shell* argument to be started instead of the default or the shell specified by the `AM_SHELL` string environment variable.

–t *tty*

This causes the UNIX *tty* device to be used for the input and output of the shell instead of the current tty.

Diagnostics

```
No writable /dev directory on Amoeba!
```

No */dev* directory could be found under Amoeba for the user. This is necessary to start a session server.

All other diagnostics come from *ax*(U) or the session server.

Environment Variables

`AM_SHELL` Specifies which shell should be started under Amoeba. The default is */bin/sh*. It is overridden by the **–s** option.

`AM_HOME` Specifies the `HOME` capability environment variable used when starting the shell. The default is */home*.

`AM_PATH` Specifies the `PATH` string environment variable used by the shell started under Amoeba. The default is */bin:/profile/util*.

`AM_TERM` This sets the UNIX terminal type which is to be used for *stdin*, *stdout* and *stderr* by the shell. The default is the UNIX `TERM` string environment variable.

Examples

In 99.99% of cases it is sufficient to type

```
amsh
```

If another shell is preferred over the default (for example */bin/ksh*) then set the `AM_SHELL` string environment variable under UNIX.

For people using X windows it is possible to set up *amsh* to be started from a menu option. For example, the following will start *amsh* in an *xterm* window if installed in a menu for *twm*.

```
"amoeba"  !"xterm -ut -geometry 80x40 -name amoeba -e amsh&"
```

See Also

ax(U), ksh(U), session(U), sh(U).

Name

amwhereis – find a file in the Amoeba distribution tree

Synopsis

```
amwhereis [-a] [-c] [-d] [-D] [-m] [-X] pattern ...
```

Description

For each regular expression *pattern* specified *amwhereis* looks for files in the Amoeba distribution whose name matches the pattern. Note that the *pattern* will be anchored to the end of the path name by *amwhereis*. Normally it searches only the Amoeba source tree but see the options below for also searching the *doc*, *conf* and *X windows* trees. It operates quickly by keeping a small set of databases of known files in `*amdir*`*/lib/whereis*. (See *amdir*(U).) Newly created files will not be reported until they have been assimilated into the database.

Note that the first time this command is run it should be run with the **–m** option to create the databases.

Options

–a Print absolute path names for matching files. Normally only paths relative to `*amdir*` are printed.

–c Search the `*amdir*`*/conf* tree for files that match the *pattern*.

–d Search the `*amdir*`*/doc* tree for files that match the *pattern*.

–D This option causes only the directory name of the matching file(s) to be printed.

–m Force a remake of the databases. *Amwhereis* remakes the database automatically in the background if it sees that the database is more than 2 days old.

–X Search the `*amdir*`*/X** tree(s) for files that match the *pattern*.

Files

`*amdir*`*/lib/amwhereis* is a directory containing the databases used by this command.

Examples

```
amwhereis /std_status.c
```

produces

```
src/lib/stubs/std/std_status.c
src/util/std/std_status.c
```

Note that it is a good idea to anchor patterns with a / where possible since it speeds the search.

```
amwhereis /std_.*s.c .*/misc/A.*
```

produces

```
src/admin/std/std_params.c
src/lib/stubs/std/std_status.c
src/util/std/std_status.c
src/lib/libam/misc/Amake.srclist
src/lib/libc/misc/Amake.srclist
src/lib/libc/machdep/mc68000/misc/Amake.srclist
src/lib/libc/machdep/generic/misc/Amake.srclist
src/lib/libc/machdep/i80386/misc/Amake.srclist
src/lib/libc/machdep/sparc/misc/Amake.srclist
src/lib/libc/machdep/Proto/misc/Amake.srclist
```

Name

aps – show process status

Synopsis

```
aps [-a] [-v] [-u username] [-l number] [-s] [directory] ...
aps [-a] [-v] [-u username] [-l number] [-s] -m [machine] ...
```

Description

Aps has two modes: *default* mode, in which *aps* uses process information retrieved from session servers, and *super* mode, in which process information is retrieved from the hosts directly. Whether super mode is available for an arbitrary user, depends on the protection of the process directories. Super mode can be specified explicitly by the **–s** option.

Aps can be invoked in two ways. The first form prints information about the processes running on each host found in each directory. If no directories are given, the default pool-processor directories are used. These processors usually contain all the processes of an ordinary user.

The second form (with **–m**) causes the remaining arguments to be interpreted as machine names instead of directories. Information about the processes of each specified machine is printed. If no machine names are given, all known machines are used.

In either form, by default, only the processes of the current user are listed (but see **–a**, below).

In default mode, *aps* prints for each of the processes managed by the session server, its process id (see below), its state, and a command string. The state is either `FREE`, `RUNNING`, `PARENT`, `CHILD`, `NEWPROC`, `EXECING`, `WAITING`, `CALLBACK`, `KILLING` or `DEAD`. When the *verbose* option is given, additionally the parent's process id, process group, status and owner (see below) are printed.

In super mode, output is columnar, and consists of a header containing column labels, followed by one line of information for each selected process. The displayed information includes, by default, the machine name (HOST), the Amoeba process table entry number (PS — the name of the entry under the *ps* directory of the machine), the status (ST — R=running, S=stopped), and the command that was used to invoke the process (COMMAND).

Note that if the HOST column shows *x* and the PS column shows *n*, the process capability can be accessed as *hostdir*/`x`/*ps*/`n`, where *hostdir* is the directory containing all the hosts of the local network (usually */super/hosts*). Among other uses, this can be supplied as an argument to *stun*(U) or *std_info*(U).

Note also that the command information is extracted from the string returned by a std_info call to the process' capability, and is not guaranteed to be correct.

If certain information is not available, an error indication is given in parentheses.

Options

–a Show all processes on the selected machines, not just the ones belonging to the current user. If specified, a USER column is added to the output, showing the user

that invoked each process. This is obtained from the std_info transaction to the process capability or to the process owner capability, and is not guaranteed to be correct.

–u *username*
Show only the processes belonging to the specified user. This option is ignored if **–a** is specified.

–l *n* Instead of printing machine name in the first column of each process line, precede the listing for each machine by a syntactically distinguished header line showing the machine name and the load on the machine (as printed by *ppload*(U)). This line shows details of available memory, CPU load and up-time. Pad the output for each machine with blank lines to at least **n** lines, to make the output of repeated calls appear at the same height. This is useful when the output is being repeatedly piped to a screen update program, e.g., by typing

```
!Gaps -al 8
```

repeatedly to *vi*.

–s Forces *aps* to go into super mode. When the **–m** or **–l** option are given, this is done automatically.

–v Add columns of information about the process that owns each listed process, obtained by doing a std_info transaction to the owner capability. The OWNER column is a string identifying the owner program (usually the session server or an *ax* server). If the process is using POSIX emulation, the POSIX PID is a unique identifier for the process, otherwise the column is blank. The POSIX PID can be supplied to the *kill*(U) command to signal the process.

If **–l** is in effect, **–v** also causes an extra line to be added to each machine header, listing the version string returned by the machine.

Examples

The command

```
aps -av
```

is executed in default mode, so it does not show any host information:

```
USER        PID  PPID  PGRP STATE      T OWNER              COMMAND
sater         2     1     2 RUNNING    R session server     -ksh
versto        2     1     2 WAITING    R session server     -ksh
versto       20     2     2 RUNNING    R session server     xload
versto       31     2     2 RUNNING    R session server     aps -av
gregor       12     1    12 RUNNING    R session server     -ksh
kaashoek     22     1    22 WAITING    R session server     -sh
kaashoek     24    22    22 RUNNING    R session server     ksh
```

The command

```
aps -av -m vmesc2 vmesc4
```

shows all processes on machines *vmesc2* and *vmesc4*:

```
           POSIX S
HOST    PS   PID T USER    OWNER              COMMAND
vmesc2   2       R condict ax server          session -a
vmesc2   4       R mjh     ax server          session -a
vmesc2   3    18 R mjh     session server     artserv -d
vmesc2   5    22 R mjh     session server     artcli
vmesc4   1       R condict ax server          run /super/cap/runsvr/
vmesc4   3    39 R sater   session server     -sh
vmesc4   5     2 R mjh     session server     -sh
vmesc4   7    19 R mjh     session server     vi temp
```

The command

```
aps -al 1 -m zoo80 zoo81
```

shows the same processes, but without the owner information and using the fixed screen format:

```
----- zoo80: load 0.00, 36.4 Mips, 25.8 Mb free, up 2 hrs 40 min --------------
  1  R chuck     session -a
----- zoo81: load 0.03, 36.4 Mips, 18.4 Mb free, up 2 hrs 41 min --------------
  1  R condict   xload -geometry 140x80-
  4  R vergo     session -a
  2  R boris     gdb grp_rf
```

It also shows the current load, theoretical CPU power available, free memory available and the uptime.

See Also

kill(U), kstat(A), ppload(U), stun(U).

Name

ascii – strip all the pure ASCII lines from a file

Synopsis

```
ascii [-n] [file]
```

Description

Sometimes a file contains some non-ASCII characters that are in the way. This program allows the lines containing only ASCII characters to be grepped from the file. With the **–n** flag, the non-ASCII lines are extracted. No matter whether the flag is used or not, the program returns an exit status of true if the file is pure ASCII, and false otherwise.

Examples

```
ascii file > outf
```

Write all the ASCII lines from *file* on *outf*.

```
ascii -n < file > outf
```

Write all the non-ASCII lines from *file* on *outf*.

Name

at – execute a command at a later date

Synopsis

```
at [options] time [date] [+ increment] [script]
```

Description

At reads commands from *script* to be executed at a later date. If *script* is omitted, commands are read from standard input. *At* will prompt for input in this case with the prompt `at>`. It then submits the script to the *sak* server (see *sak*(A)) which will execute the command script at the specified *time/date*.

The capability returned by the *sak* server will be stored in the sak directory (default */home/sak*) under an entry called at<*nr*>_*script*.job, where <*nr*> is added to create a unique filename. The name passed to the *sak* server is the basename of the specified command. This name is used by the *sak* server in status reports and info requests. The name can overridden with the **–n** option (see below).

If the **–s** option is not specified (see below) then the shell used to execute the script is taken from the current `SHELL` environment variable, if set. It defaults to */bin/sh* if the `SHELL` environment variable is not set and the **–s** option is not given. Since the *sak* server can only execute transactions, the command is packed into a transaction that will result in the following command:

`/profile/util/session -a` *shell* `-c` *script*

The current string environment is copied and used when *script* is executed. Only the `ROOT` and `WORK` capabilities are copied from the current capability environment. The **–E** and **–C** options can be used to modify the string and capability environments used to execute the command.

If no redirection is done then all output is lost.

The *time* may be specified using the following formats,

`hh:mm` `h:mm` `hhmm` `hmm` `h` `hh`

A 24 hour clock is assumed, unless the *time* is suffixed with *am* or *pm*.

Alternatively, the *time* may be specified by the special keywords *now*, *midnight* or *noon*.

An optional *date* may be specified as either a month name followed by a day number (and possibly a year), or a day of the week. Month name and day of week may be fully spelled out (in lower case) or abbreviated to three characters. Two special dates are recognized: *today* and *tomorrow*.

An optional *increment* is a number followed by *minutes, hours, days, weeks* or *years*, which can be added to a time specification. (Both singular and plural forms are accepted).

Options

–C*name=cap*	Set the capability environment variable *name* to the value *cap*.
–E 0	Execute the command with an (almost) empty string environment. The environment is actually set to contain the default values of HOME, WORK and SHELL (namely */home*, / and */bin/sh*).
–E*name=str*	Set the string environment variable *name* to the value *str*.
–c	If the *sak* server is unable to execute the job at the requested time (because the server was not running) and the **–c** flag is specified, the *sak* server will try to execute the job as soon as possible. If **–c** was not specified it will report an error.
–d *dir*	Use *dir* as sak directory (default is */home/sak*).
–n *name*	Use *name* as the name of the job when submitting it to the sak server.
–s *shell*	Use *shell* to execute *script*.

Files

/home/sak	default sak directory.
at<nr>_script.*job*	submitted *at* jobs.
sak_status	*sak* status file.
/profile/cap/saksvr/default	default *sak* server.

Examples

The following are examples of time formats and the script must be typed in as standard input.

at 3pm jan 24 1993

at 8:30 friday +1 week

at midnight

See Also

cronsubmit(U), sak(A).

Name

ax – execute program using specified environment and host

Synopsis

```
ax [options] program [argument] ...
```

Description

Ax is used to start a new Amoeba process from Amoeba or from UNIX. The file containing the executable image to be started must be installed on an Amoeba file server. (An executable file can be installed from UNIX using *ainstall*(U).)

From UNIX, *ax* is the main way to start a process running under Amoeba. When invoked, *ax* starts the process and then acts as an I/O server to handle standard input and output from the process. Two copies of the I/O server are forked, to allow simultaneous input and output. In this state, interrupt and quit signals are propagated to the process (causing its termination) or to the *session*(U) server (which should pass it on to the appropriate process group). Note that if the *ax* job dies the Amoeba process will not be able to communicate with the outside world.

Normally a shell is used to start jobs under Amoeba. When *ax* is invoked under Amoeba it is either to choose a specific processor on which to run a job or to specify a special environment. The new process inherits the `TTY`, `STDIN`, `STDOUT` and `STDERR` capabilities from *ax* (or they are redefined to some other I/O server), so *ax* does not need to provide an I/O server. Therefore *ax* exits shortly after startup, after first arranging that the new process thinks its owner is the owner of the *ax* process, and that the owner of *ax* thinks the new process is its child. This causes the shell to wait for the process started by *ax*, instead of terminating its wait when *ax* exits.

Under Amoeba, any process started by *ax* will be registered with the user's session server and its capability will appear in */dev/proc*. Processes started by *ax* under UNIX will be registered with the user's session server (if one is running) if the process makes use of the session server.

Options

–c Write the capability for the running process to standard output. In this case, all further output (e.g., the process' standard output) is redirected to the standard error output of *ax*.

–o *owner*
Sets the capability for the process owner. The owner receives a checkpoint transaction when the process exits, is stunned or causes an exception. By default, *ax* itself acts as owner, printing a traceback and terminating the process if it gets an exception or fatal signal. The traceback is printed using *pdump*(U).

–s *stacksize*
Sets the stack size in kilobytes. The default is is architecture dependent.

–m *host*
Sets the host on which the process is executed. This may either be a host name or a full

path name giving a host capability, as accepted by *host_lookup*(L). By default a suitable pool processor is chosen by *exec_findhost*(L).

−p *procsvr*
Sets the capability of the process server. This is really a different way of overriding the host on which the process is executed. It is more precise because any capability can be specified, not just a host name.

−C *name=capability*
Adds *name* to the capability environment (accessible through *getcap*(L)) and assigns it the value specified. By default the capability environment contains the following items: `STDIN`, `STDOUT`, `STDERR`: capabilities for the process' standard input, output and diagnostic output streams (implemented by *ax* acting as an I/O server). `ROOT`, `WORK`: capabilities for the process' home and work directory. They are set to the home and working directory of the user executing *ax*. `_SESSION`: capability for the session server, taken from */dev/session*, if it exists. The **−C** option can be used to override these standard items as well as to add new items.

−D *corefile*
Sets the name of the file where *ax* dumps the process descriptor if the process dies due to an exception. This option does not work when *ax* runs under the session server, because in that case the *session* server takes over the ownership of the process.

−E 0
Initialize the environment to empty. If this option is not given, the environment that *ax* inherits is passed on to the executed program. In all cases, two items are forced to standard values: `HOME` is set to / and `SHELL` is set to */bin/sh*. These, too, can be overridden using the **−E** *name=string* form below.

−E *name=string*
Adds *name* to the string environment, overriding any definition of the same name inherited from the shell or set by default.

−1 Start only a single I/O server. This is slightly more efficient if the process produces very little output, but can cause delays if it produces a lot, and makes it impossible to perform output while also waiting for input (e.g., in a different thread). By default, two I/O servers are started.

−n Do not start any I/O servers. *Ax* will exit as soon as the process is started. This is useful to start long-running servers with output redirected to a logfile using the **−C** option to redefine `STDOUT`.

−x *pathname*
Specifies the Amoeba path name for the program. By default the program name is used. If a hyphen (''−'') is specified, a capability is read from standard input. See the example below.

program
Specifies the name of the program (passed in argv[0]). Unless the **−x** option is used, this is also the Amoeba path name of the executable file.

[*argument*] ...
Optional program arguments (accessible through the normal *argc/argv* mechanism).

Environment Variables

BULLET
: This string environment variable can be set to override the name of the file server used to create the stack segment for the process (which also contains the environments and arguments).

CAPABILITY
: This environment variable is not special for *ax* but for UNIX. The string environment variable can be set to point to a UNIX file containing the capability for the host on which to run a process. It effectively defines a new root capability instead of the capability in the *.capability* file. This can be very useful if for some reason the directory server is unavailable.

Examples

To run the binary in */home/hello* on the host *bcd*:

```
ax -m bcd /home/hello
```

Sometimes under UNIX it is necessary to start programs while there is no directory server available (for example, if the *boot*(A) server fails to start for some reason.) To do this two UNIX files are need. The first (called *a.out.cap* in our example – see *ainstall*(U) for how to make this file) contains the capability of the executable. The second contains the capability for the host on which to run the program.

```
CAPABILITY=/usr/amoeba/adm/hosts/mark
BULLET=/bullet
export CAPABILITY BULLET
ax -m / -x - hello < a.out.cap
```

See Also

ainstall(U), exec_file(L), exec_findhost(L), getcap(L), getenv(L), host_lookup(L), session(U), pdump(U).

Name

banner – print a banner

Synopsis

```
banner arg ...
```

Description

Banner prints its arguments on *stdout* using a matrix of 6 x 6 pixels per character. The ‘‘@’’ sign is used for the pixels.

Example

```
banner happy birthday
```

prints a banner saying ‘‘happy birthday.’’

Name

basename – strip off file prefixes and suffixes

Synopsis

```
basename file [suffix]
```

Description

The initial directory names in the name *file* (if any) are removed yielding the name of the file itself. If a second argument is present, it is interpreted as a suffix and is also stripped, if present. This program is primarily used in shell scripts.

Examples

```
basename /bar/ast/foo
```

strips the path to yield *foo*.

```
basename /xyz/bar.c .c
```

strips the path and the *.c* to yield *bar*.

See Also

dirname(U).

Name

basic – the ACK basic compiler

Synopsis

```
basic [-o filename] file1 [file2 ...]
```

Description

Basic calls the ACK basic compiler. All the standard options of ACK work with it (see *ack*(U)). Basic source code files should have the suffix *.b*. All other files will be interpreted as either source for other languages or object files to be passed to the linker.

Options

–o filename
The output of the link editor is stored in *filename*. The default it *a.out*.

For details of other options see *ack*(U).

Diagnostics

Complaints about bad magic numbers are from the link editor and are usually the result of an incorrect file name suffix.

Warnings

All source lines must be numbered in this version of basic or else the compiler will complain.

Example

To compile the basic program in the file *prog.b* issue the command:

```
basic -o prog prog.b
```

The resultant executable will be stored in the file *prog*.

See Also

ack(U).

Name

bawk – pattern matching language

Synopsis

```
bawk rules [file] ...
```

Description

Awk is a pattern matching language. *Bawk* is Basic Awk, a subset of the original. The language is described in the *tools* section of the *Programming Guide*. The file name ‘‘–’’ can be used to designate *stdin*.

Examples

```
bawk rules input
```

Process *input* according to *rules*.

```
bawk rules - > out
```

Input from terminal, output to *out*.

Name

bsize – report the size of a Bullet file

Synopsis

```
bsize file [file2 ...]
```

Description

For each file name on the command line *bsize* prints the name of the file followed by a colon followed by a space, followed by the size of the file in bytes, followed by a newline.

Example

```
bsize /bin/cat /bin/echo
```

prints

```
/bin/cat: 49376
/bin/echo: 41184
```

Name

c2a – convert capabilities to ar-format and vice versa

Synopsis

```
a2c ar newcap
c2a [ -v ] capability...
```

Description

In some instances it is necessary to deal with capabilities as human readable strings. To do this a special format has been devised known as *ar*-format. It is described in *ar*(L).

A2c coverts *ar*-format to a capability and stores it as *newcap* using the directory primitives, or when *newcap* is a dash (‘‘-’’) it is written to standard output.

C2a prints each capability in ar-format on a separate line. The **–v** flag tells *c2a* to prefix each line with the name of the capability, and a tab.

Diagnostics

The error messages of these programs should be self-explanatory. The exit status is 2 in case of an illegal command line. *A2c* will exit with 1 if it cannot append the new capability. *C2a* will exit with 1 if it cannot lookup the capability.

Warning

Since the ar-format contains parentheses, which are special to the shell, the first argument to *a2c* must be quoted.

Examples

To save the capability *foo* in readable form on a file *foo.cap*, type:

```
c2a foo > foo.cap
```

To restore the very same capability, issue:

```
a2c "`cat foo.cap`" foo
```

See Also

ar(L).

Name

cal – print a calendar

Synopsis

```
cal [month] year
```

Description

Cal prints a calendar for a month or year. The year can be between 1 and 9999. Note that the year 91 is not a synonym for 1991, but is itself a valid year about 19 centuries ago. The calendar produced is the one used by England and her colonies. Try Sept. 1752, Feb 1900, and Feb 2000. If it is not clear what is going on, look up Calendar, Gregorian in a good encyclopedia.

Example

```
cal 3 1992
```

Prints the following calendar for March 1992.

```
      Mar 1992

 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Name

cat – concatenate files and write them to *stdout*

Synopsis

```
cat [-u] file ...
```

Description

Cat concatenates its input files and copies the result to *stdout*. If no input file is named, or ‘‘–’’ is encountered as a file name, standard input is used. Output is buffered in 512 byte blocks unless the **–u** flag is given.

Options

–u Use unbuffered output.

Examples

```
cat file
```

Display file on the terminal.

```
cat file1 file2 | lpr
```

Concatenate 2 files and print result.

Name

cc – ACK STD C Compiler

Synopsis

```
cc [options] [file ...]
```

Description

Cc is a driver that calls the STD C front end and Amsterdam Compiler Kit (ACK) back ends (see *ack*(U)) to translate a C program into ACK format object code or an Amoeba binary. By default, *cc* generates code for the architecture of the compiler itself. The program *cc* is actually a link to the generic driver of ACK *ack*(U). The only difference is that it provides a C run-time system (consisting of *libajax.a* and *libamoeba.a*) when it is used to link object files (see *led*). Here, we will only describe the options applicable when compiling C programs.

Options

–A[*file*]
: if *file* is not given, generate a list of *amake*(U) dependencies, consisting of a list of all included files, and write them to the standard output. If *file* is given, the dependencies will be generated on file *file*. Note that the dependency generation takes place in addition to compilation itself.

–D*name=text*
: define *name* as a macro with *text* as its replacement text.

–D*name*
: equivalent to **–D***name*=1.

–I*dirname*
: insert *dirname* in the list of include directories.

–M[*file*]
: generate dependencies, as with the **–A** option, only this time generate them in a format suitable as *make*(U) dependencies, i.e., of the form

    src.o: header.h

  where *src.o* is derived from the source file name.

–O
: call the target optimizer after generating assembly code.

–O*num*
: invoke the EM global optimizer before generating assembly code. The integer argument *num* determines the type and amount of optimizations being performed. It can range from 1 to 4; the higher the number the more optimization phases.

–U*name*
: undefine the previously defined macro *name*.

–a
: suppress warnings and strict standard conformance checking.

–c
: generate an object file rather than a runnable program. The default name for the object file derived from *source.c* is *source.o*. The default name for a runnable

program is *a.out*.

−c.*suffix*
like **−c**, but generate stop when a file with extension *suffix* has been produced. So **−c.s** causes *cc* to produce an assembly file. (See *ack*(U) for all file types known to ACK.)

−i when generating dependencies, do not include files from */usr/include*.

−o *name*
let the output be called *name* instead of the default *source.o*, or *a.out*.

−p generate code at each procedure entry to call the routine *procentry()*, and at each return to call the routine *procexit()*. These routines are supplied with one parameter, a pointer to a string containing the name of the procedure. This is normally used for profiling.

−s suppress strict standard conformance warnings.

−w suppress all warning messages. By default, only warning messages complaining about old-style (but legal STD C) constructions are suppressed.

−wn do not suppress warnings and stricts about old-style C, such as non-prototyped function definitions.

Refer to the ACK STD C front end documentation for a specification of the implementation-defined behavior of the ACK STD C compiler.

Diagnostics

All warning and error messages are written on standard error output.

Files

*/tmp/** – temporary files

Warnings

Although the C front end running on Amoeba is derived from the standard conforming ACK C compiler, the Amoeba headers and libraries are not all STD C conforming, because they are partly adaptations from a pre-standard C library. Some standard library functions are missing, while others will have slightly different semantics.

See Also

ack(U), amake(U), make(U),
The Link Editor in the Programming Languages section of the Programming Guide.

Name

cdiff – context diff

Synopsis

```
cdiff [-cN] old new
```

Description

Cdiff produces a context diff by first running *diff* to find the differences between the files *old* and *new* and then adding context. Some update programs, like patch, can use context diffs to update files, even in the presence of other, independent changes.

Options

–c*N* provide *N* lines of context. The default is 3.

Examples

```
cdiff old new > f
```

Write context diff on *f*.

```
cdiff -c1 old new > f
```

Use only 1 line of context.

See Also

cmp(U), diff(U), treecmp(U).

Name

cgrep – grep and display context

Synopsis

```
cgrep [-a n] [-b n] [-l n] [-w n] pattern [file] ...
```

Description

Cgrep is a program like grep, except that it also can print a few lines above and/or below the matching lines. It also prints the line numbers of the output.

Options

–a The number of lines to display after the matching line.

–b The number of lines to display before the matching line.

–f Suppress file name in the output.

–l Lines are truncated to this length before comparison.

–n Suppress line numbers in the output.

–w Sets window size. (This is equivalent to **–a***n* **–b***n*).

Example

```
cgrep -w 3 hello file1
```

Print 3 lines of context each way.

See Also

fgrep(U), grep(U).

Name

chm – change the rights masks in a directory entry

Synopsis

```
chm mask pathname ...
```

Description

Chm uses *sp_chmod*(L) to change the rights masks to *mask* in the directory entry specified by *pathname*. More than one pathname may be given. The *mask* argument should be specified as a colon-separated list of rights masks, one for each column of the directory. Each rights mask is specified by one or two hexadecimal digits. The number of mask arguments must be the same as the number of rights-mask columns in the directory entry. Otherwise the error message *invalid argument* will be given. (Use: `dir -lc pathname` to see how many rights-mask columns there are in each directory entry.)

If the path name cannot be found in the directory server or insufficient rights are available for the change-mode operation, a suitable error message will be printed.

Example

Assuming there are three rights columns, the first for ‘‘owner’’, the second for ‘‘group’’ and the third for ‘‘world’’,

```
chm ff:2:0 /home/temp
```

will set the masks so that the owner rights are not restricted (beyond the restrictions imposed by the rights in the capability), the group rights are all turned off except for the second bit (which can be on if it is present in the capability) and the world has no rights, regardless of the rights in the capability.

See Also

del(U), dir(U), get(U), mkd(U), put(U).

Name

chpw – change password

Synopsis

```
chpw [environ-file]
```

Description

Chpw sets the password that must be typed to *login*(A) (or *xlogin*) in order to log in to Amoeba. Note that only the first `MAX_PASSWD_SZ` characters (typically 8 – see *ampolicy.h*) of the password are used. Any extra characters are ignored. An encrypted version of the password is kept in the environment file (typically */Environ*). *Chpw* begins by asking for the current password. If there is none then simply press *return*. Note that as passwords are typed, they do not appear on the screen. If the current password is correctly entered *chpw* then asks for the new password. It then asks that the new password be typed again. If a different password is typed, it reports a 'Mismatch' and prompts for a retry. It exits on end-of-file. Otherwise it changes the encrypted password in the file to an encrypted version of the new password.

The optional argument *environ-file* is the name of the file that *chpw* must change. The default is the user's own environment file (namely */Environ*).

Diagnostics

Some frequent error messages:

`Mismatch`. The same password was not entered the second time.

`Operation not permitted`. The output file cannot be written.

Warnings

Because the encrypted password is stored in a file writable by the user, others with access to a user's keyboard or with write access to a user's root directory may change the password using an editor to delete the old password. This will be fixed in a future version of Amoeba.

Examples

To change one's own password:

```
chpw
```

To change Mike's password (it is necessary to be a privileged user for this):

```
chpw /super/users/mike/Environ
```

File Format

Chpw does not care much about the format of the input file. It looks for a line beginning with the word `passwd`, surrounded by white space, and replaces the rest of the line with the encrypted password, inserting a tab if necessary.

See Also

login(A).

Name

clear – clear the screen

Synopsis

```
clear
```

Description

All text is removed from the screen, resulting in an empty screen with the cursor positioned in the upper left-hand corner.

Example

```
clear
```

Clears the screen.

Name

cmp – compare two files

Synopsis

```
cmp [-ls] file1 file2
```

Description

The two files *file1* and *file2* are compared. If they are identical, exit status 0 is returned. If they differ, exit status 1 is returned. If either of the files cannot be opened, exit status 2 is returned. If *file1* is ‘‘–’’, *stdin* is compared with *file2*.

Unlike *diff*(U), this program is can used for comparing files containing binary data.

Options

–l Loud mode. Print bytes that differ (in octal)

–s Silent mode. Print nothing, just return the exit status

Examples

```
cmp file1 file2
```

Report whether the files are the same.

```
cmp -l file1 file2
```

Print all corresponding bytes that differ.

See Also

cdiff(U), diff(U), treecmp(U).

Name

comm – print lines common to two sorted files

Synopsis

```
comm [-123] file1 file2
```

Description

Two sorted files are read and compared. A three column listing is produced. Lines only in *file1* are in column 1, lines only in *file2* are in column 2 and lines common to both files are in column 3. The file name ‘‘–’’ means *stdin*.

Options

–1 Suppress column 1 (lines only in file1).

–2 Suppress column 2 (lines only in file2).

–3 Suppress column 3 (lines in both files).

Examples

```
comm file1 file2
```

Print all three columns.

```
comm -12 file1 file2
```

Print only lines common to both files.

Name

compress – compress a file using modified Lempel-Ziv coding

Synopsis

```
compress   [-cdfv] [file] ...
uncompress [file.Z] ...
zcat       [file.Z] ...
```

Description

The listed files (or *stdin*, if none are given) are compressed using the Lempel-Ziv algorithm. If the output is smaller than the input, the output is put on *file.Z* or *stdout* if no files are listed. If *compress* is linked to *uncompress*, the latter is the same as giving the **–d** flag. It will produce the file with the same name but without the *.Z* suffix and delete the original compressed file. Similarly, a link to *zcat* decompresses to *stdout*, but the original *.Z* file is not deleted.

Options

–c Put output on *stdout* instead of on *file.Z*.

–d Decompress instead of compress.

–f Force output even if there is no saving.

–v Verbose mode.

Examples

```
compress < infile > outfile
```

Compress *infile* and writes the result on *outfile*.

```
compress x y z
```

Compress the 3 files to *x.Z*, *y.Z* and *z.Z* respectively.

```
compress -d file.Z
```

Decompress *file.Z* to file.

```
uncompress file.Z
```

Equivalent to the previous example.

```
zcat file.Z
```

Decompresses *file.Z* and writes the result to *stdout*, leaving *file.Z* unchanged.

Name

cp – copy file

Synopsis

```
cp file1 file2
cp file ... directory
```

Description

Cp copies one file to another, or copies one or more files to a directory. A file cannot be copied to itself.

Examples

```
cp oldfile newfile
```

Copies *oldfile* to *newfile*.

```
cp file1 file2 /home/ast
```

Copies two files to a directory.

Name

cpdir – copy a directory and its subdirectories

Synopsis

```
cpdir [-mv] srcdir destdir
```

Description

Cpdir creates the target directory *destdir*, goes into it, and recursively copies all the files and subdirectories from the source directory *srcdir* to it. When it is finished, the target directory contains copies of files in the source directory. Capabilities for all other objects are also copied, and subdirectories are copied recursively.

Options

–v Verbose; cpdir tells what it is doing

–m Merge; just merge in the files, the target directories should already exist.

Warnings

Recursively copying subgraphs which have cycles may well loop for a long time, filling the file server's disk in the process.

Example

```
cpdir dir1 dir2
```

Create *dir2* and copy *dir1*'s subdirectories and files into it.

Name

cpp – C preprocessor

Synopsis

```
cpp [options] [inputfile]
```

Description

Cpp reads a file, expands macros and include files, and writes an input file for the C compiler. All output is to standard output. When no input file is provided as argument, *cpp* reads from standard input.

Options

–I*directory*
add this directory to the list of directories searched for

```
#include "..."
```

and

```
#include <...> commands.
```

Note that there is no space between the **–I** and the directory string. More than one **–I** option is permitted.

–I end the list of directories to be searched, and also do not look in default places.

–D*name*=*text*
define *name* as a macro with *text* as its replacement text.

–D*name*
equivalent to **–D***name*=1.

–U*name*
undefine the macro name *name*.

–C leave comments in. By default, C-comments are deleted.

–P do not generate line directives

–M*n* set maximum identifier length to *n*.

–d*file* generate a list of *amake*(U) dependencies, consisting of a list of all included files, and write them to *file*.

–i when generating dependencies, do not include files from */usr/include*.

–m when generating dependencies, generate them in the format suitable as *make*(U) dependencies, i.e.,

```
file.o: file1.h
```

where *file.o* is derived from the source file name.

–undef
this flag is silently ignored, for compatibility with other preprocessors.

The following names are always available unless undefined:

__FILE__ The input (or #include) file being compiled (as a quoted string).

__LINE__ The line number being compiled.

__DATE__ The date and time of compilation as a *ctime()* quoted string (the trailing newline is removed).

Warnings

The *cpp* as supplied is not an STD C preprocessor. As people have been known to (ab)use the C preprocessor for all kinds of weird things, this is perhaps just for the best. The ACK STD C Compiler has its own preprocessor built-in.

See Also

ack(U), amake(U), cc(U).

Name

cronsubmit – submit a crontab file to the SAK server

Synopsis

```
cronsubmit [options]
```

Description

Cronsubmit reads the file *crontab* in the sak directory (default */home/sak*) and submits it to the *sak*(A) server. A *crontab* file lists commands that are to be executed at specified times, dates or intervals.

Before submitting the jobs to the *sak* server, all existing *crontab* jobs in the sak directory will be deleted. (However see note for the **–f** option.) For each job, the capability returned by the *sak* server will be stored in the file *crontab*<*nr*>.job, where <*nr*> is the line number of the job in the crontab file. This name is also passed to the *sak* server and is used in status reports and `STD_INFO` requests.

All commands will be executed using */bin/sh –c*. Since the *sak* server can only execute transactions, the command is packed into a transaction that will result in the following command

`/profile/util/session -a` */bin/sh* `-c` *command*

Cronsubmit supplies a default capability and string environment.

The default capability environment is:

ROOT is copied from the current `ROOT` capability environment variable.

WORK is set to the `ROOT` capability environment variable.

The default string environment is:

PATH */bin:/profile/util*

_WORK /

HOME */home*

SHELL */bin/sh*

Any changes to the string or capability environment can be made using the **–E** and **–C** options.

If no redirection is done all output is lost.

Options

–Cname=cap Set the capability environment variable *name* to the value *cap*.

–E 0 Execute the command with an empty string environment.

–Ename=*str* Set the string environment variable *name* to the value *str*.

–d *dir* Use *dir* as sak directory (default */home/sak*).

–f *filename* Set the crontab file to *filename* (default filename is *crontab*). Note that when starting, only the existing cron jobs for the specified *filename* are

deleted. Those for other file names are not deleted.

Files

/home/sak	default sak directory.
crontab	default crontab file.
crontab<nr>*.job*	submitted cron job file.
/home/sak/sak_status	*sak* status file.
/profile/cap/saksvr/default	default *sak* server.

File Formats

Each line in the crontab file specifies one job and consists of six fields separated by tabs or spaces. Note that the *command* may contain further spaces without causing any confusion for *cronsubmit*. Each line has the form:

```
minutes hours   day-of-month    month   day-of-week    command
```

The respective fields have the following domains:

minutes	0 - 59.
hours	0 - 23.
day-of-month	1 - 31.
month	1 - 12.
day-of-week	0 - 6 (Sunday is day 0).
command	command to be run.

In addition any of the first five fields can be,

1) A list of values separated by commas.
2) A range specified as two numbers separated by a hyphen.
3) An asterisk (*), meaning all possible values of the field

Specification of days can be made two ways (day-of-month/month and day-of-week). If both are specified as a list of values, only one has to match for the command to be run. For example,

```
0 0 1 * 0 command
```

would run *command* on the first of every month as well as on every Sunday. To specify days by only one value the other must be set to *. For example,

```
0 0 1 * * command
```

would run *command* only on the first of every month.

See Also

at(U), sak(A), session(U), sh(U).

Name

ctags – generates ‘‘tags’’ and (optionally) ‘‘refs’’ files

Synopsis

```
ctags [-r] files...
```

Description

Ctags generates the *tags* and *refs* files from a group of C source files. The *tags* file is used by *Elvis’* ‘‘:tag’’ command, CTRL–] command, and **–t** option. The ‘‘refs’’ file is used by the *ref*(U) program.

Each C source file is scanned for `#define` statements and global function definitions. The name of the macro or function becomes the name of a tag. For each tag, a line is added to the *tags* file which contains:

- the name of the tag
- a tab character
- the name of the file containing the tag
- a tab character
- a way to find the particular line within the file.

The *refs* file is used by the *ref*(U) program, which can be invoked via the **K** command in *elvis*. When *ctags* finds a global function definition, it copies the function header into the *refs* file. The first line is flushed against the right margin, but the argument definitions are indented. The *ref* program can search the *refs* file *much* faster than it could search all of the C source files.

Options

–r This causes *ctags* to generate both *tags* and *refs*. Without **–r**, it would only generate *tags*.

Warnings

This version of ctags does not parse STD C source code very well. It has trouble recognizing the new-style function definitions.

Files

tags The ‘‘tags’’ file.

refs The ‘‘refs’’ file.

Example

The file names list will typically be the names of all C source files in the current directory.

```
ctags -r *.[ch]
```

See Also

elvis(U), ref(U).

Name

date – show date and time

Synopsis

```
date [–u] [–c] [–n] [–d dsttype] [–t minutes-west]
      [-g] [–s todserver]
      [–a [+|-]sss.fff] [+format] [[yyyy]mmddhhmm[yy][.ss]]
```

Description

This utility is part of the *localtime* package distributed though USENET.

Date without arguments writes the date and time to the standard output in the form

```
Thu Feb  8 13:51:09 MET 1990
```

with **MET** replaced by the local timezone's abbreviation (or by the abbreviation for the timezone specified in the **TZ** environment variable if set).

If a command-line argument starts with a plus sign (' + '), the rest of the argument is used as a *format* that controls what appears in the output. In the format, when a percent sign (' % ') appears, it and the character after it are not output, but rather identify part of the date or time to be output in a particular way (or identify a special character to output):

	Sample output	Explanation
%a	Wed	Abbreviated weekday name
%A	Wednesday	Full weekday name
%b	Mar	Abbreviated month name
%B	March	Full month name
%c	03/08/89 14:54:40	Month/day/year Hour:minute:second
%C	Wed Mar 8 14:54:40 1989	a la *asctime*(3)
%d	08	Day of month (always two digits)
%D	03/08/89	Month/day/year (eight characters)
%e	8	Day of month (leading zero blanked)
%h	Mar	Abbreviated month name
%H	14	24-hour-clock hour (two digits)
%I	02	12-hour-clock hour (two digits)
%j	067	Julian day number (three digits)
%k	2	12-hour-clock hour (leading zero blanked)
%l	14	24-hour-clock hour (leading zero blanked)
%m	03	Month number (two digits)
%M	54	Minute (two digits)
%n	\n	newline character
%p	PM	AM/PM designation
%r	02:54:40 PM	Hour:minute:second AM/PM designation
%R	14:54	Hour:minute
%S	40	Second (two digits)
%t	\t	tab character
%T	14:54:40	Hour:minute:second
%U	10	Sunday-based week number (two digits)

%w	3	Day number (one digit, Sunday is 0)
%W	10	Monday-based week number (two digits)
%x	03/08/89	Month/day/year (eight characters)
%X	14:54:40	Hour:minute:second
%y	89	Last two digits of year
%Y	1989	Year in full
%Z	EST	Time zone abbreviation

If a character other than one of those shown above appears after a percent sign in the format, that following character is output. All other characters in the format are copied unchanged to the output; a newline character is always added at the end of the output.

In Sunday-based week numbering, the first Sunday of the year begins week 1; days preceding it are part of ‘‘week 0.’’ In Monday-based week numbering, the first Monday of the year begins week 1.

To set the date, use a command line argument with one of the following forms:

1454	24-hour-clock hours (first two digits) and minutes
081454	Month day (first two digits), hours, and minutes
03081454	Month (two digits, January is 01), month day, hours, minutes
8903081454	Year, month, month day, hours, minutes
0308145489	Month, month day, hours, minutes, year (on System V-compatible systems)
030814541989	Month, month day, hours, minutes, four-digit year
198903081454	Four-digit year, month, month day, hours, minutes

If the century, year, month, or month day is not given, the current value is used. Any of the above forms may be followed by a period and two digits that give the seconds part of the new time; if no seconds are given, zero is assumed.

Options

−u or **−c**
Use GMT when setting and showing the date and time.

−n
Do not notify other networked systems of the time change.

−d dsttype
Set the kernel-stored Daylight Saving Time type to the given value. (The kernel-stored DST type is used mostly by ‘‘old’’ binaries.)

−t minutes-west
Set the kernel-stored ‘‘minutes west of GMT’’ value to the one given on the command line. (The kernel-stored DST type is used mostly by ‘‘old’’ binaries.)

−a adjustment
Change the time forward (or backward) by the number of seconds (and fractions thereof) specified in the *adjustment* argument. Either the seconds part or the fractions part of the argument (but not both) may be omitted. On BSD-based systems, the adjustment is made by changing the rate at which time advances; on System-V-based systems, the adjustment is made by changing the time.

–s todserver
: Specifies which time of day server has to be used. Otherwise the default (typically */profile/cap/todsvr/default*) is taken.

–g
: Only meaningful in combination with the **–s** flag. It is used to synchronize clocks. It *gets* the current time from the default time of day server and uses that to set the time on the server specified by the **–s** option. If no **–s** option is specified the default server will be set; a rather pointless thing to do.

Example

To synchronize the time of day server on machine bullet2 with the default time of day server, the following command can be used:

```
date -a+0 -g -s /super/hosts/bullet2/tod
```

If the clock on bullet2 has to be adjusted one minute, use the following command:

```
date -a+60 -s /super/hosts/bullet2/tod
```

See Also

ctime(L), posix(L), stdc(L), zdump(A), zic(A).

Name

dd – disk dumper

Synopsis

```
dd [option=value] ...
```

Description

This command is intended for copying partial files. The block size, skip count, and number of blocks to copy can be specified. The options can be used to specify other sizes than the default ones. Where sizes are expected, they are in bytes. However, the letters **w**, **b** or **k** may be appended to the number to indicate words (2 bytes), blocks (512 bytes), or K (1024 bytes), respectively. When *dd* is finished, it reports the number of full and partial blocks read and written.

Options

if=*file* Input file (default is *stdin)*.

of=*file* Output file (default is standard output).

ibs=*n* Input block size (default 512 bytes).

obs=*n* Output block size (default is 512 bytes).

bs=*n* Block size; sets ibs and obs (default is 512 bytes).

skip=*n* Skip *n* input blocks before reading.

seek=*n* Skip *n* output blocks before writing.

count=*n* Copy only n input blocks.

conv=lcase Convert upper case letters to lower case.

conv=ucase Convert lower case letters to upper case.

conv=swab Swap every pair of bytes.

conv=noerror Ignore errors and just keep going.

conv=sync Pad every input record to ibs.

conv=*arg*,*arg*[,...]
Several of the above conversions can be used simultaneously, simply by comma-separating them.

Examples

```
dd if=x of=y bs=1w skip=4
```

Copy x to y, skipping 4 words.

```
dd if=x of=y count=3
```

Copy three 512-byte blocks.

See Also

dread(A), dwrite(A).

Name

del – delete a directory entry specified by name

Synopsis

```
del [-df] pathname...
```

Description

For each specified *pathname*, *del* uses *sp_delete*(L) to delete the specified directory entry. Normally, it does not destroy any of the objects referred to by the capabilities in the capability-set of the directory entry. Thus any other directory entries referring to the same object(s) are still valid. If there are no other directory entries for one of the objects, it will eventually be destroyed by the garbage-collection process. (This is the usual way for objects to be destroyed, rather than by an explicit user command, since users do not always know whether there are multiple entries for an object.)

If any path name cannot be found in the directory server a suitable error message will be printed.

Options

–d To avoid accidents, this flag must be specified when the directory entry refers to a directory, or when the type of object cannot be determined because its server is not responding. This option forces the deletion of the directory entry but does not destroy the object referred to by the directory entry. Note that non-empty directories will be destroyed by this option.

–f This flag will cause an attempt to destroy each object referred to by the capabilities in the capability-set of each directory entry. Whether or not the attempt is fully successful, the directory entry itself will be deleted. For an object to be successfully destroyed it is necessary that the capabilities hold sufficient rights for the operation. Also, the directory server rejects attempts to destroy a non-empty directory, regardless of the capability.

Example

```
get /home/oldname | put /home/test/newname
del -d /home/oldname
```

will rename directory */home/oldname* to */home/test/newname*.

See Also

chm(U), dir(U), get(U), mkd(U), put(U).

Name

diff – print differences between two files

Synopsis

```
diff file1 file2
```

Description

Diff compares two files and generates a list of lines telling how the two files differ. Lines may not be longer than 128 characters.

Example

```
diff file1 file2
```

Print differences between *file1* and *file2*.

See Also

cdiff(U), cmp(U), treecmp(U).

Name

dir – list information about directory entries

Synopsis

```
dir [-cdklrtux] [pathname ...]
```

Description

For each *pathname*, *dir* displays on standard output, information about the specified directory entry, such as rights masks and creation time, and (optionally) about the object referred to by the entry. If no *pathname* is specified it defaults to the current working directory. If *pathname* refers to a directory, the entries in that directory are listed in alphabetic name order, rather than listing the entry that refers to the directory (but see **–d**, below).

If the path name cannot be found in the directory server or insufficient rights are available, a suitable error message will be printed.

Options

–c List the rights masks of each directory entry. The listing begins with a header line in this case. Above the file name(s) are the rights for *delete* and *modify* rights. If either is present it will appear between braces. Then for each column of the directory is printed a heading over the column showing the rights masks per directory entry. The heading printed consists of the name of each column (typically *owner group other*). If read permission for the column is present then an ‘‘*’’ will appear immediately after the column name.

–d If the object is a directory, list information about the entry for the directory object, rather than listing each entry in the directory.

–k Print, in a human-readable form, an approximation of the capability in the directory entry. This form is useful only for certain debugging purposes and cannot be translated back into a valid capability.

–l Use *std_info*(L) to obtain an informational string describing the object from the server that owns it. If the object is a directory, this string will show the rights on the directory (after the rights-masking operation is applied).

–r When listing a directory (in the absence of **–d**), sort the directory entries in reverse alphabetic name order, or the reverse of whatever order is specified by the other flags.

–t When listing a directory (in the absence of **–d**), sort the directory entries by their creation date, from newest to oldest.

–u When listing a directory (in the absence of **–d**), use the actual order that the entries appear in the directory, rather than alphabetic order.

–x Print three numbers, separated by commas, describing the capability-set in the directory entry: the initial member of the suite, the number of current members and the final member of the suite.

Certain *std_info* strings have been encoded to increase the amount of information that can be contained in them. For example, the *std_info* string for a Bullet file begins with the character ‘‘–’’ followed by the size of the file in bytes. The *std_info* string for a Soap directory begins with a ‘‘/’’ followed by the rights present for that directory. The ‘‘d’’ stands for *delete* permission, the ‘‘w’’ stands for *write* permission (i.e., modify) and the digits refer to the columns of the directory in which read permission is granted. If any of the positions in the string has a ‘‘–’’ character then that right is not present. For kernel directories the information is the same but the initial character identifying the object type is a ‘‘%’’ instead of a ‘‘/’’. For details of other encoded strings see *std_info*(U).

Warnings

Note that if the **–d** flag has *not* been specified and *pathname* refers to an object for which no server is available, then an attempt will be made to locate the server which will result in a time-out of several seconds.

Example

```
dir -lc /home/foo lib/bar
```

might produce the following output:

```
{delete|modify} owner*   group*   other*
/home/foo    ffffffff ffffffff ffffffff  Mar  9 00:31  -   362 bytes
                owner    group*   other*
lib/bar            ff       ff       ff  Mar 27 17:36  - 73952 bytes
```

The {delete|modify} above */home/foo* says that *delete* and *modify* permission is granted for */home*. This is not present for the listing of *lib/bar* so this permission is not present for the directory *lib*. An ‘‘*’’ adjacent to a column name in the header lines implies that read permission for that column is present. The OR of the rights masks in the columns for which read permission is granted is used to restrict the capability’s rights before returning it to the user.

See Also

chm(U), del(U), get(U), getdelput(U), mkd(U), put(U).

Name

dirname – print the directory component of a path name

Synopsis

```
dirname path
```

Description

Dirname is often useful in shell scripts. It takes a path name and prints all of it up to, but excluding, the last slash (‘‘/’’). If the path contains no slash character it prints ‘‘.’’ (dot).

Example

```
dirname home/foo/bar
```

will print

```
home/foo
```

and

```
dirname bar
```

will print

```
.
```

See Also

basename(U).

Name

du – show disk usage per directory or file

Synopsis

```
du [-alnosv] [-f filesvr] [path ...]
```

Description

For each directory given as argument, *du* recursively shows the number of kilobytes in use by the files. The file size information of individual files is only printed when the file is given as argument, or when the **–a** option is given. If no arguments are provided, *du* shows the disk usage of the current directory and its subdirectories.

The output of *du* is columnar, one column for each file server (see the examples section below). The file servers available are looked up in directory `/profile/cap/bulletsvr`.

Options

–a Also print the size of each individual file.

–f *fcap* Add a column for the file server with capability fcap.

–l Restrict the search of the directory graph to the local soap server.

–n For each directory, only print the total size of the files within that directory, i.e., do not add the file sizes from its subdirectories.

–o Add a column accumulating size information for objects on other file servers.

–s Do not print information about the subdirectories, only the grand total.

–v Add a column displaying for each directory the number of files and subdirectories it contains, recursively. In combination with the **–n** option, for each directory only its own contents are printed.

Examples

The command

```
du modules
```

might print something like

```
bullet0|bullet1|bullet2|
     26|     26|       | modules/h
     17|     17|       | modules/pkg
    259|    259|       | modules/lib/m68020
      |       |       | modules/lib/i80386
    259|    259|       | modules/lib
    302|    302|       | modules
```

In this example, bullet0 and bullet1 are replicated bullet file servers, hence the similarity between the first and second column.

The command

```
du -nv modules
```

will show the file sizes, number of files, and number of subdirectories for each directory:

```
bullet0|bullet1|bullet2|#fil|#sub|
     26|     26|       |  11|   0| modules/h
     17|     17|       |   4|   0| modules/pkg
    259|    259|       |  14|   0| modules/lib/m68020
       |       |       |   0|   0| modules/lib/i80386
       |       |       |   0|   2| modules/lib
       |       |       |   0|   3| modules
```

See Also

bsize(U).

Name

echo – print the arguments

Synopsis

```
echo [-n] argument ...
```

Description

Echo writes its arguments to standard output. They are separated by blanks and, unless the **–n** option is present, terminated with a line feed, This command is used mostly in shell scripts. It is available as a built-in command in most shells.

Examples

```
echo Start Phase 1
```

‘‘Start Phase 1’’ is printed.

```
echo -n Hello
```

‘‘Hello’’ is printed without a line feed.

Name

ed – editor

Synopsis

```
ed file
```

Description

Ed is a line-based editor. It is functionally equivalent to the standard V7 editor, *ed*. It supports the following commands:

(.)a: append
(.,.)c: change
(.,.)d: delete
e: edit new file
f: print name of edited file
(1,$)g: global command
(.) i: insert
(.,.+1)j: join lines together
(.)k: mark
(.)l: print with special characters in octal
(.,.)m: move
(.,.)p: print
q: quit editor
(.) r: read in new file
(.,.)s: substitute
(1,$)v: like g, except select lines that do not match
(1,$)w: write out edited file

Many of the commands can take one or two addresses, as indicated above. The defaults are shown in parentheses. Thus ‘‘a’’ appends to the current line, and ‘‘g’’ works on the whole file as default. The dot refers to the current line. Below is a sample editing session.

Examples

```
ed prog.c
```

Edit *prog.c*.

```
3,20p
```

Print lines 3 through 20.

```
/whole/
```

Find next occurrence of whole.

```
s/whole/while/
```

Replace ‘‘whole’’ by ‘‘while.’’

```
g/Buf/s//BUF/g
```

Replace ‘‘Buf’’ by ‘‘BUF’’ everywhere.

```
w
```

Write the file back.

```
q
```

Exit the editor.

Note

Ed is part of the third party software.

Name

eject – eject the disk from a removable-disk drive

Synopsis

```
eject disk-cap
```

Description

This command is used to eject floppies and CDs from devices that can only eject their media under software control. This is typically true for Sun 4 machines.

It will not eject disks that can only be ejected manually, such as on most PCs.

Diagnostics

If the disk specified is not software ejectable or the user has insufficient permission then an error message will be printed. No error is printed if no disk was present in the drive.

Example

```
eject /super/hosts/jumbo/floppy:00
```

will eject the floppy disk from drive 0 on the host *jumbo*.

See Also

vdisk(A).

Name

elvis – full screen editor

Synopsis

```
elvis [flags] [+cmd] [files...]
vi    [flags] [+cmd] [files...]
ex    [flags] [+cmd] [files...]
view  [flags] [+cmd] [files...]
```

Description

Elvis is a text editor which emulates *vi* and *ex*.

On Amoeba, *elvis* should also be installed under the names *ex*, *vi* and *view*. These extra names would normally be links to elvis; see the *ln*(U) command.

When *elvis* is invoked as *vi*, it behaves exactly as though it was invoked as *elvis*. However, if *elvis* is invoked as *view*, then the read-only option is set as though the **–R** flag was given. If *elvis* is invoked as *ex*, then *elvis* will start up in the colon command mode instead of the visual command mode, as though the **–e** flag had been given. If *elvis* is invoked as *edit*, then *elvis* will start up in input mode, as though the **–i** flag was given.

Options

–r To the real *vi*, this flag means that a previous edit should be recovered. *Elvis*, though, has a separate program, called *elvrec*(U), for recovering files. When invoked with **–r**, *elvis* will tell you to run *elvrec*.

–R This sets the ‘‘read-only’’ option, to prevent accidental overwriting of a file.

–t *tag* This causes *elvis* to start editing at the given tag.

–m [*file*] *Elvis* will search through *file* for something that looks like an error message from a compiler. It will then begin editing the source file that caused the error, with the cursor sitting on the line where the error was detected. If a *file* is not explicitly named then *errlist* is assumed.

–e *Elvis* will start up in colon command mode.

–v *Elvis* will start up in visual command mode.

–i *Elvis* will start up in input mode.

+*command* After the first file is loaded *command* is executed as an *ex* command. A typical example would be

```
elvis +237 foo
```

which would cause *elvis* to start editing *foo* and then move directly to line 237.

Files

/tmp/elv* During editing, *elvis* stores text in a temporary file. For Amoeba, this file will usually be stored in the */tmp* directory, and the first three characters will be ‘‘elv’’. For other systems, the temporary files may be stored someplace else; see the version-specific section of the documentation.

tags This is the database used by the *:tags* command and the **–t** option. It is usually created by the *ctags*(U) program.

Warnings

There is no LISP support. Certain other features are missing, too.

Auto-indent mode is not quite compatible with the real *vi*. Among other things, 0^D and ^^D are different.

Long lines are displayed differently. The real *vi* wraps long lines onto multiple rows of the screen, but *elvis* scrolls sideways.

See Also

ctags(U), ref(U), elvprsv(U). elvrec(U).

Name

elvprsv – preserve the the modified version of a file after elvis crashes

Synopsis

```
elvprsv ["-why reason"] /tmp/filename ...
elvprsv -R /tmp/filename ...
```

Description

If *elvis* crashes for some reason after making unsaved changes to a file then it calls *elvprsv* to preserve the changes. The changes can be recovered later using the *elvrec* program.

Elvprsv should never need to be run from the command line. It is run automatically when *elvis* is about to die.

If editing a file when *elvis* dies (due to a bug, system crash, power failure, etc.), *elvprsv* will preserve the most recent version of the file. The preserved text is stored in a special directory; it does NOT overwrite the original file automatically.

If the program *mail* has been installed then *elvprsv* will send mail to any user whose work it preserves.

If a nameless buffer was being edited when *elvis* died, then *elvrec* will pretend that the file was named *foo*.

Files

*/tmp/elv**

The temporary file that *elvis* was using when it died.

*/dev/dead/p**

The text that is preserved by *elvprsv*.

/dev/dead/ElvisIndex

A text file which lists the names of all preserved files, and the names of the */dev/dead/p** files which contain their preserved text.

See Also

elvis(U), elvrec(U).

Name

elvrec – recover the modified version of a file after a crash

Synopsis

elvrec [preservedfile [newfile]]

Description

If editing a file with *elvis*(U) and it dies due to a signal, the most recent version of the file being edited will be preserved. The preserved text is stored in a special directory; it does NOT overwrite the original text file automatically.

Elvrec locates the preserved version of a given file using the *preservedfile* argument. It must be exactly the same file name as given to the original aborted *elvis* command. *Elvrec* writes the recovered version over the top of the original file – or to a new file, if the *newfile* argument was specified. The recovered file will have *nearly* all of the changes.

To see a list of all recoverable files, run *elvrec* with no arguments.

If a nameless buffer was being edited when *elvis* died, then *elvrec* will pretend that the file was named *foo*.

Files

*/dev/dead/p**

The text that was preserved when *elvis* died.

/dev/dead/ElvisIndex

A text file which lists the names of all preserved files, and the corresponding names of the */dev/dead/p** files which contain the preserved text.

Warnings

Elvrec is very picky about filenames. Exactly the same path name given to *elvis* must be specified. The simplest way to do this is to go into the same directory where *elvis* was running, and invoke *elvrec* with no arguments. This will give exactly which path name was used for the desired file.

See Also

elvis(U), elvprsv(U).

Name

envcap – get a capability from, or list, the capability environment

Synopsis

```
envcap [name]
```

Description

With no arguments, *envcap* lists the names of all the capabilities in the capability environment.

Given the name of one of the capabilities in the environment, *envcap* uses *getcap*(L) to obtain the capability specified by *name* and writes it to standard output, in the form of a singleton capability-set suitable as input to the *put* command. No check for the validity of the capabilities in the environment is performed.

Example

```
envcap TTY | put /dev/tty34
```

will install the `TTY` capability of the current process under the name */dev/tty34* in the directory server. If any process then writes to */dev/tty34*, the output will go to the tty that was in use when the envcap command was invoked.

See Also

get(U), getcap(L), put(U).

Name

expand – convert tabs to spaces

Synopsis

```
expand [-t1,t2, ...] [file]
```

Description

Expand replaces tabs in the named files with the equivalent numbers of spaces. If no files are listed, *stdin* is used. The default is a tab every 8 spaces.

Options

–*n* Tab stop positions at the *n* th character position. If only one tab position is given, the rest are multiples of it.

Example

```
expand -16,32,48,64
```

Expand *stdin* with tabs every 16 columns.

See Also

unexpand(U).

Name

expr – evaluate expression

Synopsis

```
expr expression operator expression ...
```

Description

Expr computes the value of its arguments and writes the result on standard output. The valid operators, in order of increasing precedence, are listed below. Grouped operators have the same precedence.

|
: Or: return the left hand side if it is neither 0 nor the empty string, otherwise return the right hand side.

&
: And: return the left hand side if neither left or right hand side are 0 or the empty string. Otherwise it returns 0.

<, <=, ==, !=, >=, >
: If both expressions are integer, return the result of the numeric comparison, otherwise do a lexical comparison.

+, –
: Add or subtract two integer values.

*
: Multiply two integer values.

:
: Match the first argument to the second, the latter being interpreted as regular expression. If the regular expression contains a ‘‘\(regexp\)’’ pair, the matched part is returned, otherwise it evaluates to the number of characters matched.

Example

```
x=`expr $x + 1`
```

Add 1 to shell variable *x*.

```
x=`expr $path : '.*/\(.*\)'`
```

If *$path* is a path name containing one or more ‘‘/’’ characters, *x* will be set to the last component.

Name

f2c – convert FORTRAN 77 to C or C++

Synopsis

```
f2c [options] files ...
```

Description

F2c converts FORTRAN 77 source code in *files* with names ending in *.f* or *.F* to C (or C++) source files in the current directory, with *.c* substituted for the final *.f* or *.F*. If no FORTRAN files are named, *f2c* reads FORTRAN from standard input and writes C on standard output. Filenames that end with *.p* or *.P* are taken to be prototype files, as produced by option **–P** (see below), and are read first.

The resulting C invokes the support routines of *f77*; object code should be loaded with ACK (see *f77*(U) and *ack*(U)).

Note that unless special options provided by *f2c* are required, it is simpler to call *f77* directly to compile FORTRAN sources since this produces binaries directly.

Options

The following are standard *f77* options.

–C Compile code to check that subscripts are within declared array bounds.

–I2 Render `INTEGER` and `LOGICAL` as short, `INTEGER*4` as *long int*. Assume the default library *tail_f77*: allow only `INTEGER*4` (and no `LOGICAL`) variables in `INQUIRE`s.

–I4 confirms the default rendering of `INTEGER` as *long int*.

–onetrip
Compile DO loops that are performed at least once if reached. (FORTRAN 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)

–U The case of variable and external names is significant. FORTRAN keywords must be in *lower case*.

–u Make the default type of a variable ‘undefined’ rather than using the default FORTRAN rules.

–w Suppress all warning messages.

–w66 Only FORTRAN 66 compatibility warnings are suppressed.

The following options are peculiar to *f2c*.

–A Produce STD C. Default is old-style C.

–a Make local variables automatic rather than static unless they appear in a `DATA`, `EQUIVALENCE`, `NAMELIST`, or `SAVE` statement.

–C++ Produce C++ code.

–c Include original FORTRAN source as comments.

–E Declare uninitialized `COMMON` to be `Extern` (overridably defined in *f2c.h* as `extern`).

–ec Place uninitialized `COMMON` blocks in separate files. For example, `COMMON /ABC/` appears in the file *abc_com.c*.

–e1c This bundles the separate files into the output file, with comments that give an unbundling *sed*(U) script.

–ext Complain about *f77* extensions.

–g Include original FORTRAN line numbers as comments.

–h Try to align character strings on word boundaries.

–hd Try to align character strings on double-word boundaries.

–i2 Similar to **–I2**, but assume a modified *tail_f77* (compiled with **–Df2c_i2**), so `INTEGER` and `LOGICAL` variables may be assigned by `INQUIRE` and array lengths are stored in short ints.

–kr Use temporary values to enforce FORTRAN expression evaluation where K&R (first edition) parenthesization rules allow rearrangement.

–krd Use double precision temporaries even for single-precision operands.

–P Create a *file***.P** of STD C (or C++) prototypes for procedures defined in each input *file***.f** or *file***.F.** When reading FORTRAN from standard input, write prototypes at the beginning of standard output. Implies **–A** unless option **–C++** is present.

–Ps This implies **–P**, and gives exit status 4 if rerunning *f2c* may change prototypes or declarations.

–p Supply preprocessor definitions to make common-block members look like local variables.

–R Do not promote `REAL` functions and operations to `DOUBLE PRECISION`.

–!R confirms the default, which imitates *f77*.

–r Cast values of `REAL` functions (including intrinsics) to `REAL`.

–r8 Promote `REAL` to `DOUBLE PRECISION`, `COMPLEX` to `DOUBLE COMPLEX`.

–T dir Put temporary files in directory *dir*.

–w8 Suppress warnings when `COMMON` or `EQUIVALENCE` forces odd-word alignment of doubles.

–W n Assume *n* characters/word (default 4) when initializing numeric variables with character data.

–z Do not implicitly recognize `DOUBLE COMPLEX`.

–!bs Do not recognize *b*acks*l*ash escapes (\", \', \0, \\, \b, \f, \n, \r, \t, \v) in character strings.

–!c Inhibit C output, but produce **–P** output.

–!I Reject **include** statements.

–!it Do not infer types of untyped `EXTERNAL` procedures from use as parameters to previously defined or prototyped procedures.

–!P Do not attempt to infer STD C or C++ prototypes from usage.

Files

**.f* and **.F*: FORTRAN source files.
**.c*: C source files produced by f2c.
/profile/module/ack/include/_tail_cc/f2c.h is the include file used by the C code produced by *f2c*.
*/profile/module/ack/lib/*ARCH*/tail_f77*: FORTRAN run-time libraries loaded by *f77*.

Warnings

Floating-point constant expressions are simplified in the floating-point arithmetic of the machine running *f2c*, so they are typically accurate to at most 16 or 17 decimal places.

Untypable EXTERNAL functions are declared *int*.

See Also

ack(U), f77(U), sed(U).

Name

f77 – the FORTRAN compiler driver

Synopsis

```
f77 [-c] [-o filename] file1 [file2 ...]
```

Description

The FORTRAN compiler driver *f77* first converts FORTRAN sources provided to C using *f2c*(U). If no problems are encountered, the generated source code is translated to object code using the ACK ANSI compiler *cc*(U).

Fortran source files should have the suffix *.f*. All other files will be interpreted as either source for other languages or object files. When no **–c** option is specified, *f77* will link the object files together with a FORTRAN-specific library.

Options

–c Only produce object files; do not call the linker.

–o filename

The output of the link editor is stored in *filename*. The default is *a.out*.

As *f77* is really a link to the generic compiler driver *ack*, it will also accept several other options. Refer to *ack*(U) for the details.

Diagnostics

While processing a FORTRAN source, the program *f2c* will print the names of the functions it encounters on standard output.

Example

To compile the FORTRAN program in the file *prog.f* issue the command:

```
f77 -o prog prog.f
```

The resultant executable will be stored in the file *prog*.

See Also

f2c(U), ack(U).

Name

factor – factor an integer less than 2^{32}

Synopsis

```
factor number
```

Description

Factor prints the prime factors of its argument in increasing order. Each factor is printed as many times as it appears in the number.

Example

```
factor 450180
```

Print the prime factors of 450180, i.e.,

```
2 2 3 3 5 41 61
```

Name

false – exit with the value false

Synopsis

```
false
```

Description

This command returns the value false. It is used for shell programming.

See Also

true(U).

Name

fgrep – fast grep

Synopsis

```
fgrep [-chlnsv] [[-e special-string] | [-f file] | string] [file] ...
```

Description

Fgrep is essentially the same as *grep*, except that it only searches for lines containing literal *string*s (no wildcard characters), and it is faster.

If no *file* names are given it searches standard input. The following options modify the behavior of *fgrep*.

Options

–c Count matching lines and only print count, not the lines.

–e *special-string*
Used when the string to search for begins with a ‘‘-’’ character.

–f *file*
Take strings from file named in following argument.

–h Omit file headers from printout.

–l List file names once only.

–n Each line is preceded by its line number.

–s Status only, no output.

–v Print only lines not matching.

Examples

```
fgrep % prog.c
```

Print lines containing a % sign.

```
fgrep -f pattern prog.c
```

Print all the lines in *prog.c* that contain lines matching the string in the file *pattern*.

See Also

cgrep(U), grep(U).

Name

file – make a guess as to a file's type based on contents

Synopsis

```
file name ...
```

Description

File reads the first block of a file and tries to make an intelligent guess about what kind of file it is. It understands about archives, C source programs, loadable objects, Amoeba binaries, shell scripts, and English text.

Example

```
file a.out ar.h src
```

This might print:

```
a.out: mc68000 executable
ar.h: C program
src: directory
```

Name

find – walk a file hierarchy

Synopsis

```
find [-dsXx] [-f file] [file ...] expression
```

Description

Find recursively descends the directory tree for each *file* listed, evaluating an *expression* (composed of the ‘‘primaries’’ and ‘‘operands’’ listed below) in terms of each file in the tree. This program was ported from BSD and contains many options that are of no use under Amoeba.

Options

The options are as follows:

–d The **–d** option causes *find* to perform a depth–first traversal, i.e., directories are visited in post–order and all entries in a directory will be acted on before the directory itself. By default, *find* visits directories in pre–order, i.e., before their contents. Note, the default is *not* a breadth–first traversal.

–f The **–f** option specifies a file hierarchy for *find* to traverse. File hierarchies may also be specified as the operands immediately following the options.

–s The **–s** option is redundant under Amoeba.

–X The **–X** option is a modification to permit *find* to be safely used in conjunction with *xargs*(U). If a file name contains any of the delimiting characters used by *xargs*, a diagnostic message is displayed on standard error, and the file is skipped. The delimiting characters include single (‘‘ ’ ’’) and double (‘‘ " ’’) quotes, backslash (‘‘\’’), space, tab and newline characters.

–x The **–x** option is redundant under Amoeba.

Primaries

All primaries which take a numeric argument allow the number to be preceded by a plus sign (‘‘+’’) or a minus sign (‘‘–’’). A preceding plus sign means ‘‘more than n’’, a preceding minus sign means ‘‘less than n’’ and neither means ‘‘exactly n’’ .

Note that Amoeba only maintains a modification date with files so **–atime** and **–ctime** are equivalent to **–mtime**.

–atime *n* True if the difference between the file last access time and the time *find* was started, rounded up to the next full 24–hour period, is *n* 24–hour periods.

–ctime *n* True if the difference between the time of last change of file status information and the time *find* was started, rounded up to the next full 24–hour period, is *n* 24–hour periods.

–exec *utility* [*argument ...*]*;*
True if the program named *utility* returns a zero value as its exit status. Optional arguments may be passed to the utility. The expression must be terminated by a

semicolon (‘‘;’’). If the string ‘‘{}’’ appears anywhere in the utility name or the arguments it is replaced by the path name of the current file. *Utility* will be executed from the directory from which *find* was executed.

–fstype *type*

True if the file is contained in a file system of type *type*. Currently supported types are ‘‘local’’, ‘‘mfs’’, ‘‘nfs’’, ‘‘msdos’’, ‘‘isofs’’, ‘‘fdesc’’, ‘‘kernfs’’, ‘‘rdonly’’ and ‘‘ufs’’. The types ‘‘local’’ and ‘‘rdonly’’ are not specific file system types. The former matches any file system physically mounted on the system where the *find* is being executed and the latter matches any file system which is mounted read-only.
Amoeba currently only supports one file system type so this primary is redundant.

–group *gname*

True if the file belongs to the group *gname*. If *gname* is numeric and there is no such group name, then *gname* is treated as a group id.
Amoeba currently only supports a single group so this primary is redundant.

–inum *n* True if the file has inode number *n*.

–links *n* True if the file has *n* links.
Amoeba does not keep link counts. However, it does provide links.

–ls This primary always evaluates to true. The following information for the current file is written to standard output: its inode number, size in 512–byte blocks, file permissions, number of hard links, owner, group, size in bytes, last modification time, and path name. If the file is a block or character special file, the major and minor numbers will be displayed instead of the size in bytes. If the file is a symbolic link, the path name of the linked–to file will be displayed preceded by ‘‘–>’’. The format is identical to that produced by ‘‘ls –dgils’’.

–mtime *n* True if the difference between the file last modification time and the time *find* was started, rounded up to the next full 24–hour period, is *n* 24–hour periods.

–ok *utility* [*argument* ...]*;*

The **–ok** primary is identical to the **–exec** primary with the exception that *find* requests user affirmation for the execution of the utility by printing a message to the terminal and reading a response. If the response is other than ‘‘y’’ the command is not executed and the value of the *ok* expression is false.

–name *pattern*

True if the last component of the path name being examined matches *pattern*. Special shell pattern matching characters (‘‘[’’, ‘‘]’’, ‘‘*’’, and ‘‘?’’) may be used as part of *pattern*. These characters may be matched explicitly by escaping them with a backslash (‘‘\’’).

–newer *file*

True if the current file has a more recent last modification time than *file*.

–nouser True if the file belongs to an unknown user. Amoeba files are ‘‘ownerless’’ in the POSIX sense and so this primary always evaluates to true.

–nogroup True if the file belongs to an unknown group. Amoeba files are ‘‘groupless’’ in the POSIX sense and so this primary always evaluates to true.

–path *pattern*

True if the path name being examined matches *pattern*. Special shell pattern matching characters (‘‘[’’, ‘‘]’’, ‘‘*’’, and ‘‘?’’) may be used as part of *pattern*. These characters may be matched explicitly by escaping them with a backslash (‘‘\’’). Slashes (‘‘/’’) are treated as normal characters and do not have to be matched explicitly.

−perm [−*mode*]

The *mode* may be either symbolic (see *chmod*(U)) or an octal number. If the mode is symbolic, a starting value of zero is assumed and the mode sets or clears permissions without regard to the process’ file mode creation mask. If the mode is octal, only bits 07777 (`S_ISUID | S_ISGID | S_ISTXT | S_IRWXU | S_IRWXG | S_IRWXO`) of the file’s mode bits participate in the comparison. If the mode is preceded by a dash (‘‘−’’), this primary evaluates to true if at least all of the bits in the mode are set in the file’s mode bits. If the mode is not preceded by a dash, this primary evaluates to true if the bits in the mode exactly match the file’s mode bits. Note, the first character of a symbolic mode may not be a dash (‘‘−’’).

−print This primary always evaluates to true. It prints the path name of the current file to standard output. The expression is appended to the user specified expression if neither **−exec**, **−ls** or **−ok** is specified.

−prune This primary always evaluates to true. It causes *find* to not descend into the current file. Note, the **−prune** primary has no effect if the **−d** option was specified.

−size *n*[*c*] True if the file’s size, rounded up, in 512−byte blocks is *n*. If *n* is followed by a ‘‘c’’, then the primary is true if the file’s size is *n* bytes.

−type *t* True if the file is of the specified type. Possible file types are as follows:

- b block special
- c character special
- d directory
- f regular file
- l symbolic link
- p FIFO
- s socket

Note that Amoeba does not support sockets or symbolic links. Nor does it directly support concepts like block special and character special devices. However the POSIX emulation will report servers and other, for POSIX unrecognizable objects, as block or character special devices.

−user *uname*

True if the file belongs to the user *uname*. If *uname* is numeric and there is no such user name, then *uname* is treated as a user id.
Since Amoeba does not support POSIX style file ownership, all files belong to the user *anybody*. Therefore this option is redundant.

Operators

The primaries may be combined using the following operators. The operators are listed in order of decreasing precedence.

(*expression*) This evaluates to true if the parenthesized expression evaluates to true.

! *expression* This is the unary NOT operator. It evaluates to true if the expression is false.

expression **–and** *expression*

expression expression

The **–and** operator is the logical AND operator. As it is implied by the juxtaposition of two expressions; it does not have to be specified. The expression evaluates to true if both expressions are true. The second expression is not evaluated if the first expression is false.

expression **–or** *expression*

The **–or** operator is the logical OR operator. The expression evaluates to true if either the first or the second expression is true. The second expression is not evaluated if the first expression is true.

All operands and primaries must be separate arguments to *find*. Primaries which themselves take arguments expect each argument to be a separate argument to *find*.

Warnings

The special characters used by *find* are also special characters to many shell programs. In particular, the characters ‘‘*’’, ‘‘[’’, ‘‘]’’, ‘‘?’’, ‘‘(’’, ‘‘)’’, ‘‘!’’, ‘‘\’’ and ‘‘;’’ may have to be escaped from the shell.

As there is no delimiter separating options and file names or file names and the *expression*, it is difficult to specify files named ‘‘-xdev’’ or ‘‘!’’. These problems are handled by the **–f** option and the *getopt* ‘‘--’’ construct.

Find does its best to avoid cycles in the directory graph but is not foolproof.

POSIX Conformance

The *find* utility syntax is a superset of the syntax specified by the POSIX 1003.2 standard.

The **–X** option and the **–inum** and **–ls** primaries are extensions to POSIX 1003.2 .

Historically, the **–d, –s** and **–x** options were implemented using the primaries ‘‘–depth’’, ‘‘–follow’’, and ‘‘–xdev’’. These primaries always evaluated to true. As they were really global variables that took effect before the traversal began, some legal expressions could have unexpected results. An example is the expression ‘‘–print –o –depth’’. As **–print** always evaluates to true, the standard order of evaluation implies that **–depth** would never be evaluated. This is not the case.

The operator **–or** was implemented as **–o**, and the operator **–and** was implemented as **–a**.

Historic implementations of the **–exec** and **–ok** primaries did not replace the string ‘‘{}’’ in the utility name or the utility arguments if it had preceding or following non-whitespace characters. This version replaces it no matter where in the utility name or arguments it appears.

Examples

The following examples are shown as given to the shell:

```
find / \! -name *.c -print
```

Print out a list of all the files whose names do not end in ‘‘.c’’.

```
find / -newer ttt -user wnj -print
```

Print out a list of all the files owned by user ‘‘wnj’’ that are newer than the file ‘‘ttt’’.

```
find / \! \( -newer ttt -user wnj \) -print
```

Print out a list of all the files which are not both newer than ‘‘ttt’’ and owned by ‘‘wnj’’.

```
find / \( -newer ttt -or -user wnj \) -print
```

Print out a list of all the files that are either owned by ‘‘wnj’’ or that are newer than ‘‘ttt’’.

See Also

starch(U).

Name

flex – fast lexical analyzer generator

Synopsis

```
flex [-dfirstvFILT -c[efmF] -Sskeleton_file] [filename]
```

Description

Flex is a rewrite of *lex* intended to right some of that tool's deficiencies: in particular, *flex* generates lexical analyzers much faster, and the analyzers use smaller tables and run faster.

Options

–d This makes the generated scanner run in *debug* mode. Whenever a pattern is recognized the scanner will write to *stderr* a line of the form:

```
--accepting rule #n
```

Rules are numbered sequentially with the first one being 1.

–f This has the same effect as *lex*'s **–f** flag (do not compress the scanner tables); the mnemonic changes from *fast compilation* to either *full table* or *fast scanner*. The actual compilation takes *longer*, since *flex* is I/O bound writing out the big table. This option is equivalent to **–cf** (see below).

–i This instructs *flex* to generate a *case-insensitive* scanner. The case of letters given in the *flex* input patterns will be ignored, and the rules will be matched regardless of case. The matched text given in *yytext* will have the preserved case (i.e., it will not be folded).

–r specifies that the scanner uses the **REJECT** action.

–s causes the *default rule* (that unmatched scanner input is echoed to *stdout*) to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

–t puts the result on standard output instead of the file *lex.yy.c*.

–v has the same meaning as for *lex* (print to *stderr* a summary of statistics of the generated scanner). Many more statistics are printed, though, and the summary spans several lines. Most of the statistics are meaningless to the casual *flex* user.

–F specifies that the *fast* scanner table representation should be used. This representation is about as fast as the full table representation **(–f)**, and for some sets of patterns will be considerably smaller (and for others, larger). In general, if the pattern set contains both ‘‘keywords’’ and a catch-all, ‘‘identifier’’ rule, such as in the set:

```
"case"    return ( TOK_CASE );
"switch"  return ( TOK_SWITCH );
"default" return ( TOK_DEFAULT );
[a-z]+    return ( TOK_ID );
```

then it is better using the full table representation. If only the identifier rule is

present, and a hash table or some such thing is used to detect the keywords, it is better to use **–F**.
This option is equivalent to **–cF** (see below).

–I instructs *flex* to generate an *interactive* scanner. Normally, scanners generated by *flex* always look ahead one character before deciding that a rule has been matched. At the possible cost of some scanning overhead (it is not clear that more overhead is involved), *flex* will generate a scanner which only looks ahead when needed. Such scanners are called *interactive* because to write a scanner for an interactive system such as a command shell, it will probably need the user's input to be terminated with a newline, and without **–I** the user will have to type a character in addition to the newline in order to have the newline recognized. This leads to dreadful interactive performance.

If all this seems to confusing, here is the general rule: if a human will be typing in input to a scanner, use **–I,** otherwise do not; if it does not matter how fast the scanner runs and it is not necessary to make any assumptions about the input to the scanner, always use **–I**.

Note, **–I** cannot be used in conjunction with *full* or *fast tables*, i.e., the **–f**, **–F**, **–cf**, or **–cF** flags.

–L instructs *flex* to not generate *#line* directives (see below).

–T makes *flex* run in *trace* mode. It will generate a lot of messages to standard out concerning the form of the input and the resultant non-deterministic and deterministic finite automatons. This option is mostly for use in maintaining *flex*.

–c[efmF] controls the degree of table compression. **–ce** directs *flex* to construct *equivalence classes*, i.e., sets of characters which have identical lexical properties (for example, if the only appearance of digits in the *flex* input is in the character class [0-9] then the digits 0, 1, ..., 9 will all be put in the same equivalence class). **–cf** specifies that the *full* scanner tables should be generated – *flex* should not compress the tables by taking advantages of similar transition functions for different states. **–cF** specifies that the alternate fast scanner representation (described above under the **–F** flag) should be used. **–cm** directs *flex* to construct *meta-equivalence classes*, which are sets of equivalence classes (or characters, if equivalence classes are not being used) that are commonly used together. A lone **–c** specifies that the scanner tables should be compressed but neither equivalence classes nor meta-equivalence classes should be used.

The options **–cf** or **–cF** and **–cm** do not make sense together – there is no opportunity for meta-equivalence classes if the table is not being compressed. Otherwise the options may be freely mixed.

The default setting is **–cem** which specifies that *flex* should generate equivalence classes and meta-equivalence classes. This setting provides the highest degree of table compression. One can trade off faster-executing scanners at the cost of larger tables with the following generally being true:

```
slowest             smallest
          -cem
          -ce
          -cm
          -c
          -c{f,F}e
          -c{f,F}
fastest             largest
```

–Sskeleton_file overrides the default skeleton file from which *flex* constructs its scanners. This option will not be needed unless doing *flex* maintenance or development.

Incompatibilities With Lex

flex is fully compatible with *lex* with the following exceptions:

- There is no run-time library to link with. It is not necessary to specify **–ll** when linking, and a main program must be supplied. (Hacker's note: since the *lex* library contains a *main()* which simply calls *yylex()*, it is possible to be lazy and not supply a main program and link with **–ll**.)
- *lex*'s *%r* (Ratfor scanners) and *%t* (translation table) options are not supported.
- The do-nothing **–n** flag is not supported.
- When definitions are expanded, *flex* encloses them in parentheses. With *lex*, the following

  ```
  NAME    [A-Z][A-Z0-9]*
  %%
  foo{NAME}?      printf("Found it\n");
  %%
  ```

  will not match the string ''foo'' because when the macro is expanded the rule is equivalent to ''foo[A-Z][A-Z0-9]*?'' and the precedence is such that the ''?'' is associated with ''[A-Z0-9]*''. With *flex*, the rule will be expanded to ''foo([A-z][A-Z0-9]*)?'' and so the string ''foo'' will match.
- *yymore* is not supported.
- The undocumented *lex*-scanner internal variable *yylineno* is not supported.
- If the input uses *REJECT, flex* must be run with the **–r** flag. If it is omitted, the scanner will abort at run-time with a message that the scanner was compiled without the flag being specified.
- The *input()* routine is not redefinable, though may be called to read characters following whatever has been matched by a rule. If *input()* encounters and end-of-file the normal *yywrap()* processing is done. A ''real'' end-of-file is returned as *EOF*.

  Input can be controlled by redefining the YY_INPUT macro. YY_INPUT's calling sequence is

  ```
  YY_INPUT(buf, result, max_size)
  ```

Its action is to place up to *max_size* characters in the character buffer *buf* and return in the integer variable *result* either the number of characters read or the constant YY_NULL (0 on UNIX systems) to indicate EOF. The default YY_INPUT reads from the file-pointer *yyin* (which is by default *stdin*), so if a change to the input file is required, it is not necessary to redefine YY_INPUT – just point *yyin* at the input file.

A sample redefinition of YY_INPUT (in the first section of the input file):

```
%{
#undef YY_INPUT
#define YY_INPUT(buf, result, max_size) \
    result = (buf[0] = getchar()) == EOF ? YY_NULL : 1;
%}
```

Things like keeping track of the input line number can be added this way; but do not expect the scanner to go very fast.

- *output* is not supported. Output from the ECHO macro is done to the file-pointer *yyout* (default *stdout*).
- Trailing context is restricted to patterns which have either a fixed-sized leading part or a fixed-sized trailing part. For example, ‘‘a*/b’’ and ‘‘a/b*’’ are okay, but not ‘‘a*/b*’’. This restriction is due to a bug in the trailing context algorithm given in *Principles of Compiler Design* (and *Compilers – Principles, Techniques, and Tools*) which can result in mismatches. Try the following *lex* program

```
%%
x+/xy           printf("I found \"%s\"\n", yytext);
```

on the input ‘‘xxy’’. If an arbitrary trailing context is unavoidable, *yyless* can be used to effect it.

- *flex* reads only one input file, while *lex*’s input is made up of the concatenation of its input files.

Enhancements

- *Exclusive start-conditions* can be declared by using *%x* instead of *%s*. These start-conditions have the property that when they are active, *no other rules are active*. Thus a set of rules governed by the same exclusive start condition describe a scanner which is independent of any of the other rules in the *flex* input. This feature makes it easy to specify ‘‘mini-scanners’’ which scan portions of the input that are syntactically different from the rest (e.g., comments).
- *flex* dynamically resizes its internal tables, so directives like ‘‘%a 3000’’ are not needed when specifying large scanners.
- The scanning routine generated by *flex* is declared using the macro *YY_DECL*. By redefining this macro the routine’s name and calling sequence can be changed. For example, use

```
#undef YY_DECL
#define YY_DECL float lexscan( a, b ) float a, b;
```

to give it the name *lexscan*, returning a float, and taking two floats as arguments.

- *flex* generates *#line* directives mapping lines in the output to their origin in the input file.
- Multiple actions can be put on the same line, separated by semi-colons. With *lex*, the following

```
foo    handle_foo(); return 1;
```

  is truncated to

```
foo    handle_foo();
```

  flex does not truncate the action. Actions that are not enclosed in braces are terminated at the end of the line.
- Actions can be begun with %{ and terminated with %}. In this case, *flex* does not count braces to figure out where the action ends – actions are terminated by the closing *%}*. This feature is useful when the enclosed action has extraneous braces in it (usually in comments or inside inactive #ifdef's) that throw off the brace-count.
- All of the scanner actions (e.g., *ECHO*, *yywrap*...) except the *unput()* and *input()* routines, are written as macros, so they can be redefined if necessary without requiring a separate library to link to.

Diagnostics

flex scanner jammed

A scanner compiled with **–s** has encountered an input string which was not matched by any of its rules.

flex input buffer overflowed

A scanner rule matched a string long enough to overflow the scanner's internal input buffer (as large as *BUFSIZ* in ''stdio.h''). Edit *flexskelcom.h* and increase *YY_BUF_SIZE* and *YY_MAX_LINE* to increase this limit.

REJECT used and scanner was not generated using –r

Just like it sounds. The scanner uses *REJECT*. Run *flex* on the scanner description using the **–r** flag.

old-style lex command ignored

The *flex* input contains a *lex* command (e.g., ''%n 1000'') which is being ignored.

Bugs

Use of *unput()* or *input()* trashes the current *yytext* and *yyleng*.

Use of *unput()* to push back more text than was matched can result in the pushed-back text matching a beginning-of-line ('^') rule even though it did not come at the beginning of the line.

Nulls are not allowed in *flex* inputs or in the inputs to scanners generated by *flex*. Their presence generates fatal errors.

Do not mix trailing context with the '|' operator used to specify that multiple rules use the same action. That is, avoid constructs like:

```
foo/bar      |
bletch       |
bugprone     { ... }
```

They can result in subtle mismatches. This is actually not a problem if there is only one rule using trailing context and it is the first in the list (so the above example will actually work okay). The problem is due to fall-through in the action switch statement, causing non-trailing-context rules to execute the trailing-context code of their fellow rules. This should be fixed, as it is a nasty bug and not obvious. The proper fix is for *flex* to spit out a FLEX_TRAILING_CONTEXT_USED #define and then have the backup logic in a separate table which is consulted for each rule-match, rather than as part of the rule action. The place to do the tweaking is in *add_accept()* – any kind soul want to be a hero?

The pattern:

```
x{3}
```

is considered to be variable-length for the purposes of trailing context, even though it has a clear fixed length.

Due to both buffering of input and read-ahead, one cannot intermix calls to, for example, *getchar()* with *flex* rules and expect it to work. Call *input()* instead.

The total table entries listed by the **–v** flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states if the scanner was not compiled with **–r**, and greater than the number of states if it was.

The scanner run-time speeds have not been optimized as much as they deserve. Van Jacobson's work shows that the scanner can go quite a bit faster still.

Files

flex.skel skeleton scanner
flex.fastskel skeleton scanner for **–f** and **–F**
flexskelcom.h common definitions for skeleton files
flexskeldef.h definitions for compressed skeleton file
fastskeldef.h definitions for **–f**, **–F** skeleton file.

See Also

M. E. Lesk and E. Schmidt, *LEX – Lexical Analyzer Generator*.

Name

fmt – adjust line-length for paragraphs of text

Synopsis

```
fmt [-width] [files]...
```

Description

Fmt is a simple text formatter. It inserts or deletes newlines, as necessary, to make all lines in a paragraph be approximately the same width. It preserves indentation and word spacing.

The default line width is 72 characters. This can be overridden with the **–width** flag. If no files are named on the command line, *fmt* will read from *stdin*.

Example

It is typically used from within *vi* to adjust the line breaks in a single paragraph. To do this, move the cursor to the top of the paragraph, type

```
!}fmt
```

and press the return key.

From the command line it can be called to format the file *foo* to contain no lines longer than 75 characters with the command:

```
fmt -75 foo
```

Name

fold – fold long lines

Synopsis

```
fold [-n] [file] ...
```

Description

Fold copies its input from the named file (or *stdin*, if none is specified) to standard output. However, lines longer than the given maximum length (default 80) are broken into multiple lines of the maximum length by inserting new line characters.

Options

–n *n* is a decimal integer specifying how long the output lines should be.

Examples

```
fold -60
```

Fold *stdin* to 60 characters.

```
fold -1 file
```

Fold file to 80 characters.

Name

format – format a physical disk

Synopsis

```
format disk-cap
```

Description

This command is used to format disks. It is primarily intended for formatting floppy disks. Attempts to apply the command to virtual disks will fail. It should be applied only to the capability for the entire device. In the case of hard disks this is the *bootp:nn* capability. In the case of floppies it is the *floppy:nn* capability.

Unlike other systems, no progress reports are printed during formatting. It is a single command to the virtual disk server.

Warnings

Formatting disks is dangerous. It will destroy any data present on the disk.

Some disk drivers do not support formatting.

Diagnostics

The driver for the specified disk may not support formatting. In this case the error `STD_COMBAD` (bad command) will be returned. All other errors can arise due to insufficient permission or removal of the media during the formatting process.

Example

```
format /super/hosts/jumbo/floppy:00
```

will format the floppy disk in drive 0 on the host *jumbo*.

See Also

vdisk(A).

Name

fortune – print a fortune

Synopsis

```
fortune
```

Description

Fortune picks a fortune at random from the fortune file, */usr/lib/fortune.dat*, and prints it. This file consists of pieces of text separated by a line containing only %%.

Example

```
fortune
```

This might print:

```
First Law of Socio-Genetics:
Celibacy is not hereditary.
```

or

```
Structured Programming supports the law of the excluded muddle.
```

Name

fromb – read a file from the Bullet Server

Synopsis

```
fromb [-o offset] [-s size] filename
```

Description

Fromb reads the Bullet file named *filename* and writes its contents to standard output. If the file name cannot be found in the directory server or the capability is invalid a suitable error message will be printed. If the offset is greater than the file size it will print an error message and exit. If the offset is valid and *offset* plus *size* is greater than the file size it will read up until the end of the file.

Since this command works under UNIX it provides a mechanism for transferring Bullet files from Amoeba to UNIX.

Options

–o offset starts reading the file at *offset* bytes from the start of the file. The default is 0.

–s size says to read a maximum of *size* bytes. The default is the whole file.

Example

```
fromb /home/textfile | wc -l
```

will read the file */home/textfile* and pipe it through the *wc* command and print the number of lines in the file.

See Also

bsize(U), cat(U), tob(U).

Name

ftp – Internet file transfer program

Synopsis

```
ftp [-dginv] host
```

Description

Ftp is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user.

Options

Options may be specified at the command line, or to the command interpreter.

–v Verbose option forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

–n Restrains *ftp* from attempting auto-login upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc* (see below) file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will prompt for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

–i Turns off interactive prompting during multiple file transfers.

–d Enables debugging.

–g Disables file name globbing.

Commands

When *ftp* is awaiting commands from the user, the prompt

```
ftp>
```

is provided to the user. The following commands are recognized by *ftp*:

! [*command* [*args*]]
Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be a command to execute directly, with the rest of the arguments as its arguments.

$ *macro-name* [*args*]
Execute the macro *macro-name* that was defined with the **macdef** command. Arguments are passed to the macro unglobbed.

account [*passwd*]
Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is

included, the user will be prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file after being altered by any **ntrans** or **nmap** setting. File transfer uses the current settings for **type**, **format**, **mode**, and **structure**.

ascii Set the file transfer **type** to network `ASCII`. This is the default type.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer **type** to support binary image transfer.

bye Terminate the `FTP` session with the remote server and exit *ftp*. An end of file will also terminate the session and exit.

case Toggle remote computer file name case mapping during **mget** commands. When **case** is on (default is off), remote computer file names with all letters in upper case are written in the local directory with the letters mapped to lower case.

cd *remote-directory*
Change the working directory on the remote machine to *remote-directory*.

cdup Change the remote machine working directory to the parent of the current remote machine working directory.

chmod *mode file-name*
Change the permission modes of the file *file-name* on the remote system to *mode*.

close Terminate the `FTP` session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr Toggle carriage return stripping during ascii type file retrieval. Records are denoted by a carriage return/linefeed sequence during ascii type file transfer. When **cr** is on (the default), carriage returns are stripped from this sequence to conform with the single linefeed record delimiter. Records on non–UNIX remote systems may contain single linefeeds; when an ascii type transfer is made, these linefeeds may be distinguished from a record delimiter only when **cr** is off.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified it is used to set the debugging level. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string ‘‘`-->`’’.

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **dir** output. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, or *local-file* is ‘‘**–**’’, output comes to the terminal.

disconnect
A synonym for *close*.

form *format*
Set the file transfer **form** to *format*.

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine, subject to alteration by the current **case**, **ntrans**, and **nmap** settings. The current settings for **type**, **form**, **mode**, and **structure** are used while transferring the file.

glob Toggle filename expansion for **mdelete**, **mget** and **mput**. If globbing is turned off with **glob**, the file name arguments are taken literally and not expanded. Globbing for **mput** is done as in *ksh*(U). For **mdelete** and **mget**, each remote file name is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: the exact result depends on the foreign operating system and *ftp* server, and can be previewed by doing

```
mls remote-files -
```

Note: **mget** and **mput** are not meant to transfer entire directory subtrees of files. That can be done by transferring a *tar*(U) archive of the subtree (in binary mode).

hash Toggle hash-sign (‘‘#’’) printing for each data block transferred. The size of a data block is 1024 bytes.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of the known commands.

idle [*seconds*]
Set the inactivity timer on the remote server to *seconds* seconds. If *seconds* is omitted, the current inactivity timer is printed.

lcd [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user’s home directory is used.

ls [*remote-directory*] [*local-file*]
Print a listing of the contents of a directory on the remote machine. The listing includes any system-dependent information that the server chooses to include; for example, most UNIX systems will produce output from the command ‘‘`ls -l`’’. (See also **nlist**.) If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **ls** output. If no local file is specified, or if *local-file* is ‘‘**–**’’, the output is sent to the terminal.

macdef *macro-name*
Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain defined until a **close** command is executed.

The macro processor interprets '$' and '\' as special characters. A '$' followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A '$' followed by an 'i' signals that macro processor that the executing macro is to be looped. On the first pass '$i' is replaced by the first argument on the macro invocation command line, on the second pass it is replaced by the second argument, and so on. A '\' followed by any character is replaced by that character. Use the '\' to prevent special treatment of the '$'.

mdelete [*remote-files*]
Delete the *remote-files* on the remote machine.

mdir *remote-files* [*local-file*
Like **dir**, except multiple remote files may be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target *local-file* for receiving **mdir** output.

mget *remote-files*
Expand the *remote-files* on the remote machine and do a **get** for each file name thus produced. See **glob** for details on the filename expansion. Resulting file names will then be processed according to **case**, **ntrans**, and **nmap** settings. Files are transferred into the local working directory, which can be changed with the **lcd** command; new local directories can be created with

```
! mkdir directory
```

mkdir *directory-name*
Make a directory on the remote machine.

mls *remote-files local-file*
Like **nlist**, except multiple remote files may be specified, and the *local-file* must be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mls** output.

mode [*mode-name*]
Set the file transfer **mode** to *mode-name*. The default mode is `stream` mode.

modtime *file-name*
Show the last modification time of the file on the remote machine.

mput *local-files*
Expand wild cards in the list of local files given as arguments and do a **put** for each file in the resulting list. See **glob** for details of filename expansion. Resulting file names will then be processed according to **ntrans** and **nmap** settings.

newer *file-name*
Get the file only if the modification time of the remote file is more recent that the file on the current system. If the file does not exist on the current system, the remote file is considered **newer**. Otherwise, this command is identical to *get*.

nlist [*remote-directory*] [*local-file*]
Print a list of the files in a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **nlist** output. If no local file is specified, or if *local-file* is ''**–**'',

the output is sent to the terminal.

nmap [*inpattern outpattern*]

Set or unset the filename mapping mechanism. If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during **mget** commands and **get** commands issued without a specified local target filename.

This command is useful when connecting to a non−UNIX remote computer with different file naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. *Inpattern* is a template for incoming filenames (which may have already been processed according to the **ntrans** and **case** settings). Variable templating is accomplished by including the sequences '$1', '$2', ..., '$9' in *inpattern*. Use '\' to prevent this special treatment of the '$' character. All other characters are treated literally, and are used to determine the **nmap** *inpattern* variable values.

For example, given *inpattern* $1.$2 and the remote file name ''mydata.data'', $1 would have the value ''mydata'', and $2 would have the value ''data''.

The *outpattern* determines the resulting mapped filename. The sequences '$1', '$2', ..., '$9' are replaced by any value resulting from the *inpattern* template. The sequence '$0' is replace by the original filename. Additionally, the sequence [*seq1*, *seq2*] is replaced by *seq1* if *seq1* is not a null string; otherwise it is replaced by *seq2*.

For example, the command

```
nmap $1.$2.$3 [$1,$2].[$2,file]
```

would yield the output filename ''myfile.data'' for input filenames ''myfile.data'' and ''myfile.data.old'', ''myfile.file'' for the input filename ''myfile'', and ''myfile.myfile'' for the input filename ''.myfile''. Spaces may be included in *outpattern*, as in the example:

```
nmap $1 sed "s/  *$//" > $1
```

Use the '\' character to prevent special treatment of the '$', '[', ']', and ',' characters.

ntrans [*inchars* [*outchars*]]

Set or unset the filename character translation mechanism. If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during **mget** commands and **get** commands issued without a specified local target filename.

This command is useful when connecting to a non−UNIX remote computer with different file naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the

character is deleted from the file name.

open *host* [*port*]
Establish a connection to the specified *host* FTP server. An optional port number may be supplied, in which case, *ftp* will attempt to contact an FTP server at that port. If the **auto-login** option is on (default), *ftp* will also attempt to automatically log the user in to the FTP server (see below).

prompt Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default is on), any **mget** or **mput** will transfer all files, and any **mdelete** will delete all files.

proxy *ftp-command*
Execute an *ftp* command on a secondary control connection. This command allows simultaneous connection to two remote *ftp* servers for transferring files between the two servers. The first **proxy** command should be an **open**, to establish the secondary control connection. Enter the command ‘‘proxy ?’’ to see other *ftp* commands executable on the secondary connection.

The following commands behave differently when prefaced by **proxy**: **open** will not define new macros during the auto-login process, **close** will not erase existing macro definitions, **get** and **mget** transfer files from the host on the primary control connection to the host on the secondary control connection, and **put**, **mput**, and **append** transfer files from the host on the secondary control connection to the host on the primary control connection.

Third party file transfers depend upon support of the FTP protocol PASV command by the server on the secondary control connection.

put *local-file* [*remote-file*]
Store a local file on the remote machine. If *remote-file* is left unspecified, the local file name is used after processing according to any **ntrans** or **nmap** settings in naming the remote file. File transfer uses the current settings for **type**, **format**, **mode**, and **structure**.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for **bye**.

quote *arg1* [*arg2* ...]
The arguments specified are sent, verbatim, to the remote FTP server.

recv *remote-file* [*local-file*]
A synonym for get.

reget *remote-file* [*local-file*]
Reget acts like get, except that if *local-file* exists and is smaller than *remote-file*, *local-file* is presumed to be a partially transferred copy of *remote-file* and the transfer is continued from the apparent point of failure. This command is useful when transferring very large files over networks that are prone to dropping connections.

remotehelp [*command-name*]
Request help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

remotestatus [*file-name*]
With no arguments, show status of remote machine. If *file-name* is specified, show status of *file-name* on remote machine.

rename *from to*
Rename the file *from* on the remote machine, to the file *to*.

reset Clear reply queue. This command re-synchronizes command/reply sequencing with the remote *ftp* server. Resynchronization may be necessary following a violation of the FTP protocol by the remote server.

restart *marker*
Restart the immediately following **get** or **put** at the indicated *marker*. On UNIX systems, marker is usually a byte offset into the file.

rmdir *directory-name*
Delete a directory on the remote machine.

runique
Toggle storing of files on the local system with unique filenames. If a file already exists with a name equal to the target local filename for a **get** or **mget** command, a ‘‘.1’’ is appended to the name. If the resulting name matches another existing file, a ‘‘.2’’ is appended to the original name. If this process continues up to ‘‘.99’’, an error message is printed, and the transfer does not take place. The generated unique filename will be reported. Note that **runique** will not affect local files generated from a shell command (see below). The default value is off.

send *local-file* [*remote-file*]
A synonym for put.

sendport
Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, *ftp* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they have been accepted.

site *arg1* [*arg2* ...]
The arguments specified are sent, verbatim, to the remote FTP server as a SITE command.

size *file-name*
Return size of *file-name* on the remote machine.

status Show the current status of *ftp*.

struct *struct-name*
Set the file transfer *structure* to *struct-name*. By default stream structure is used.

sunique Toggle storing of files on the remote machine under unique file names. The remote FTP server must support FTP protocol STOU command for successful completion. The remote server will report a unique name. Default value is off.

system Show the type of operating system running on the remote machine.

tenex Set the file transfer type to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer **type** to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

umask [*newmask*]
Set the default umask on the remote server to *newmask*. If *newmask* is omitted, the current umask is printed.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the *password* is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If an *account* field is not specified, and the FTP server requires it, the user will be prompted for it. If an *account* field is specified, an account command will be relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless *ftp* is invoked with auto-login disabled, this process is done automatically on initial connection to the FTP server.

verbose
Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, verbose is on.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quote ‘"’ marks.

Aborting a File Transfer

To abort a file transfer, use the terminal interrupt key (usually Ctrl-C). Sending transfers will be immediately halted. Receiving transfers will be halted by sending an FTP protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server’s support for ABOR processing. If the remote server does not support the ABOR command, an ftp> prompt will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence will be ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may result from the ABOR processing described above, or from unexpected behavior by the remote server, including violations of the FTP protocol. If the delay results from unexpected remote server behavior, the local *ftp* program must be killed by hand.

File Naming Conventions

Files specified as arguments to *ftp* commands are processed according to the following rules.

(1) If the file name ‘‘–’’ is specified, then *stdin* (for reading) or *stdout* (for writing) is used.

(2) If the first character of the file name is ‘‘|’’, the remainder of the argument is

interpreted as a shell command. *Ftp* then forks a shell, using *popen* from the standard library with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; e.g. `"| ls -lt"`. A particularly useful example of this mechanism is:

```
dir -l |yap
```

(3) Failing the above checks, if ‘‘globbing’’ is enabled, local file names are expanded according to the rules used in the *ksh*(U). If the *ftp* command expects a single local file (.e.g. **put**) only the first filename generated by the ‘‘globbing’’ operation is used.

(4) For **mget** commands and **get** commands with unspecified local file names, the local filename is the remote filename, which may be altered by a **case**, **ntrans**, or **nmap** setting. The resulting filename may then be altered if **runique** is on.

(5) For **mput** commands and **put** commands with unspecified remote file names, the remote filename is the local filename, which may be altered by an **ntrans** or **nmap** setting. The resulting filename may then be altered by the remote server if **sunique** is on.

File Transfer Parameters

The FTP specification specifies many parameters which may affect a file transfer. The **type** may be one of `ascii`, `image` (binary), `ebcdic`, and `local byte size` (for PDP-10s and PDP-20s mostly). *Ftp* supports the `ascii` and `image` types of file transfer, plus local byte size 8 for **tenex** mode transfers.

Ftp supports only the default values for the remaining file transfer parameters: **mode**, **form**, and **struct**.

The .netrc File

The *.netrc* file contains login and initialization information used by the auto-login process. It resides in the user’s home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

machine *name*

Identify a remote machine *name*. The auto-login process searches the *.netrc* file for a **machine** token that matches the remote machine specified on the *ftp* command line or as an **open** command argument. Once a match is made, the subsequent *.netrc* tokens are processed, stopping when the end of file is reached or another **machine** or a **default** token is encountered.

default This is the same as **machine** *name* except that **default** matches any name. There can be only one **default** token, and it must be after all **machine** tokens. This is normally used as:

```
default login anonymous password user@site
```

thereby giving the user *automatic* anonymous FTP login to machines not specified in *.netrc*. This can be overridden by using the **–n** flag to disable auto-login.

login *name*

Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified *name*.

password *string*

Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the *.netrc* file for any user other than *anonymous*, *ftp* will abort the auto-login process if the *.netrc* is readable by anyone besides the user.

account *string*

Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password, or the auto-login process will initiate an ACCT command if it does not.

macdef *name*

Define a macro. This token functions like the *ftp* **macdef** command functions. A macro is defined with the specified name; its contents begin with the next *.netrc* line and continue until a null line (consecutive new-line characters) is encountered. If a macro named **init** is defined, it is automatically executed as the last step in the auto-login process.

Environment Variables

Ftp utilizes the following environment variables.

HOME For default location of a *.netrc* file, if one exists.

SHELL For default shell.

Warnings

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of carriage returns in the BSD 4.2 ascii-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from BSD 4.2 servers using the ascii type. Avoid this problem by using the binary image type.

See Also

ftpd(A).

Name

gax – group ax

Synopsis

```
gax [options] progname NCPU [args ...]
```

Description

In Amoeba it is possible to write programs which comprise several interdependent processes working together in parallel (for example, Orca programs). *Gax* is used to start such programs. It takes as its main argument a program to be executed and starts several copies of the program simultaneously. Typically each copy is started on a different processor, although this can be controlled with the options. The primary duties of *gax* are to select the appropriate processors upon which to run a job and then start the job running. It will continue as the parent of the job to deal with any signals or other process management for all members of the job.

Gax must be provided with the capabilities for a pool directory from which it is to select processors upon which to execute the processes that form the job. It will then poll all these processors in the manner of the run server (see *run*(A)) and determine their load and available memory. Based on this and any selection criteria specified in the options (described below) it will select the best available processors to execute the job.

Options

–1 Do not run more than one process on a processor. If more processes are required to be started than available processors in the pool (for example, if some machines have crashed) it is an error and the program will not be executed. Additionally, this option causes *gax* to delay starting the job until sufficient memory (as specified by the **–m** option) is available.

–e *host-list*
The *host-list* is a file containing the names of the processors to be used. Only processors that are both in the processor pool and in the list will be used to run processes. Processors will be selected in the order in which they appear in the list.

–m *mbytes*
Only use hosts with at least *mbytes* megabytes of free memory.

–p *pool*
Specifies that *pool* should be used as the pool directory from which to select processors. Pool directories should be structured in the same manner as *run*(A) server pools. The default pool is */profile/gaxpool*.

–r By default, *gax* examines the run server status maintained for the pool in order to skip hosts that are down. This may help speeding up process startup in case one or more hosts in the pool are down. However, when the hosts are reserved by means of the reservation service (see reserve (U)), the run server also reports them to be down, even though they are not. In this case, the information can be ignored by specifying **–r** option. Setting the `POOL` environment variable also has this effect.

–s Some systems have hosts on more than one network, with the networks being joined by a switch. This option causes hosts to be allocated so that processes are evenly *scattered* over the different network segments. The name of the hosts are used to determine the network structure. The last character of the name specifies the host id within a segment and the second last character the network segment id. For example, the host *zoo41* is processor 1 on segment 4 of the set of processors known as *zoo*. It will be necessary to use letters as well as digits if there are more than 10 hosts on a segment.

–t *timeout*
Kill all the processes after *timeout* seconds if they have not already terminated.

–v Verbose mode. Details of processor selection and process startup are printed.

Environment Variables

`POOL`
This is an alternative way to specify the pool parameter. Putting ‘‘POOL=pool’’ in the environment of *gax* has the same effect as providing it with the ‘‘-p pool -r’’ options. This feature is used by the reservation service to indicate the reserved pool to be used by *gax*.

The capability environment of the executed programs is augmented with the following capabilities:

`MEMBERCAP`
per-process put-port for communication with its ‘control’ thread.

`GROUPCAP`
port for group communication shared by all spawned processes

`MEMBER%d`
`NCPU` get-ports. Usually `MEMBER`*n* is used by process *n*. Because all processes get this list it can (also) be used for interprocess communication.

Files

/profile/gaxpool
The default pool directory.

Example

To run the Orca program *tsp* on 5 processors from the private processor pool */my_pool*:

```
gax -p /my_pool tsp 5
```

See Also

ax(U), reserve(U), run(A).

Name

get – get a capability-set specified by name

Synopsis

```
get pathname
```

Description

get uses *sp_lookup*(L) to obtain the capability-set specified by *pathname* and writes it on standard output. If the path name cannot be found in the directory server a suitable error message will be printed. No check for the validity of the capabilities in the set is performed.

Example

```
get /home/oldname | put /home/newname
```

will install */home/newname* in the directory server as a synonym for */home/oldname*.

See Also

chm(U), dir(U), getdelput(U), mkd(U), del(U), put(U).

Name

getdelput – atomically replace a capability with another

Synopsis

```
getdelput src_cap dest_cap
```

Description

The function of *getdelput* is to atomically delete the capability-set in the directory entry *dest_cap* and replace it with the capability-set in the directory entry *src_cap*. Both directory entries must exist.

Diagnostics

If *src_cap* does not exist then the message

```
getdelput: can't lookup src_cap (not found)
```

will be printed. If *dest_cap* does not exist then the message

```
getdelput: can't replace dest_cap (not found)
```

will be printed.

Example

```
getdelput file1 file2
```

will delete the capability-set in the directory entry *file2* (but not destroy the object associated with it) and replace it with the capability-set *file1*.

See Also

get(U), put(U).

Name

gprof – display call graph profile data

Synopsis

```
gprof [options] [a.out] [gmon.out]
```

Description

Gprof produces an execution profile of C programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (*gmon.out* default) which is created by programs that are compiled with the **–pg** option of *gcc*, and linked with the library *libgprof.a*. *Gprof* reads the given object file (the default is *a.out*) and establishes the relation between its symbol table and the call graph profile from *gmon.out*. If more than one profile file is specified, the *gprof* output shows the sum of the profile information in the given profile files.

Gprof calculates the amount of time spent in each routine. Next, these times are propagated along the edges of the call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle. The first listing shows the functions sorted according to the time they represent including the time of their call graph descendents. Below each function entry is shown its (direct) call graph children, and how their times are propagated to this function. A similar display above the function shows how this function's time and the time of its descendents is propagated to its (direct) call graph parents.

Cycles are also shown, with an entry for the cycle as a whole and a listing of the members of the cycle and their contributions to the time and call counts of the cycle.

Second, a flat profile is given, similar to that provided by *prof*(1) under UNIX. This listing gives the total execution times, the call counts, the time in milliseconds the call spent in the routine itself, and the time in milliseconds the call spent in the routine itself including its descendents.

Finally, an index of the function names is provided.

Options

The following options are available:

–a Suppresses the printing of statically declared functions. If this option is given, all relevant information about the static function (e.g., time samples, calls to other functions, calls from other functions) belongs to the function loaded just before the static function in the *a.out* file.

–b Suppresses the printing of a description of each field in the profile.

–c The static call graph of the program is discovered by a heuristic that examines the text space of the object file. Static-only parents or children are shown with call counts of 0.

–e *name*

Suppresses the printing of the graph profile entry for routine *name* and all its descendants (unless they have other ancestors that are not suppressed). More than

one **–e** option may be given. Only one *name* may be given with each **–e** option.

–E *name*

Suppresses the printing of the graph profile entry for routine *name* (and its descendants) as **–e**, above, and also excludes the time spent in *name* (and its descendants) from the total and percentage time computations. (For example, **–E** *mcount* **–E** *mcleanup* is the default.)

–f *name*

Prints the graph profile entry of only the specified routine *name* and its descendants. More than one **–f** option may be given. Only one *name* may be given with each **–f** option.

–F *name*

Prints the graph profile entry of only the routine *name* and its descendants (as **–f**, above) and also uses only the times of the printed routines in total time and percentage computations. More than one **–F** option may be given. Only one *name* may be given with each **–F** option. The **–F** option overrides the **–E** option.

–k *fromname toname*

Will delete any arcs from routine *fromname* to routine *toname*. This can be used to break undesired cycles. More than one **–k** option may be given. Only one pair of routine names may be given with each **–k** option.

–s A profile file *gmon.sum* is produced that represents the sum of the profile information in all the specified profile files. This summary profile file may be given to later executions of gprof (probably also with a **–s**) to accumulate profile data across several runs of an *a.out* file.

–z Displays routines that have zero usage (as shown by call counts and accumulated time). This is useful with the **–c** option for discovering which routines were never called.

Files

a.out The namelist and text space.

gmon.out Dynamic call graph and profile.

gmon.sum Summarized dynamic call graph and profile.

Warnings

The granularity of the sampling is shown, but remains statistical at best. We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called. Thus the time propagated along the call graph arcs to the function’s parents is directly proportional to the number of times that arc is traversed.

Parents that are not themselves profiled will have the time of their profiled children propagated to them, but they will appear to be spontaneously invoked in the call graph listing, and will not have their time propagated further. Similarly, signal catchers, even though profiled, will appear to be spontaneous (although for more obscure reasons). Any profiled children of signal catchers should have their times propagated properly, unless the signal

catcher was invoked during the execution of the profiling routine, in which case all is lost.

The profiled program must call *exit*() or return normally for the profiling information to be saved in the *gmon.out* file.

See Also

GCC user manual.

Name

grep – search a file for lines containing a given pattern

Synopsis

```
grep [-insv] [-e] pattern [file ...]
```

Description

Grep searches one or more files (by default, *stdin*) and selects out all the lines that match the *pattern*. All the regular expressions accepted by *ed*(U) are allowed. In addition, ‘‘+’’ can be used instead of ‘‘*’’ to mean 1 or more occurrences and ‘‘?’’ can be used to mean 0 or 1 occurrences. between two regular expressions to mean either one of them. Parentheses can be used for grouping. If a match is found, exit status 0 is returned. If no match is found, exit status 1 is returned. If an error is detected, exit status 2 is returned.

Options

–e pattern
This is the same as *pattern*. The **–e** is useful for patterns starting with ‘-’,

–i Does case-insensitive matching. That is, no distinction is made between upper and lower-case letters.

–n Print line numbers as well as the matching lines.

–s Sets the exit status only, no printed output.

–v Select lines that do not match the specified pattern.

Examples

```
grep mouse file
```

Find lines in file containing mouse.

```
grep [0-9] file
```

Print lines containing a digit.

See Also

cgrep(U), fgrep(U).

Name

head – print the first few lines of a file

Synopsis

```
head [-n] [file] ...
```

Description

The first few lines of one or more files are printed. The default count is 10 lines. The default file is *stdin*.

Options

–n Specifies how many lines to print.

Examples

```
head -6
```

Print first 6 lines of *stdin*.

```
head -1 file1 file2
```

Print first line of *file1* and the first line of *file2* plus a little context information.

See Also

tail(U).

Name

host – look up host names using domain name server

Synopsis

```
host [-l] [-v] [-w] [-r] [-d] [-t querytype] [-a] host [server]
```

Description

Host looks for information about Internet hosts. The Internet is accessed via the TCP/IP server. It gets this information from a set of interconnected servers that are spread across the world. By default, it simply converts between host names and Internet addresses. However with the **–t** or **–a** options, it can be used to find all of the information maintained by the domain server about a particular host.

The arguments can be either host names or host numbers. The program first attempts to interpret them as host numbers. If this fails, it will treat them as host names. A host number consists of first decimal numbers separated by dots, e.g., 128.6.4.194 A host name consists of names separated by dots, e.g., *topaz.rutgers.edu*. Unless the name ends in a dot, the local domain is automatically tacked on the end. Thus a Rutgers user can say `host topaz`, and it will actually look up *topaz.rutgers.edu*. If this fails, the name is tried unchanged (in this case, *topaz*). This same convention is used for mail and other network utilities. The actual suffix to tack on the end is obtained by looking at the results of a *hostname* call, and using everything starting at the first dot. (See below for a description of how to customize the host name lookup.)

The first argument is the host name to look up. If this is a number, an ‘‘inverse query’’ is done. That is, the domain system looks in a separate set of databases used to convert numbers to names.

The second argument is optional. It allows the specification of a particular server to query. If this argument is not specified the default server (normally the local machine) is used.

If a name is specified, output of three different kinds may appear. Here is an example that shows all of them:

```
% host sun4
sun4.rutgers.edu is a nickname for ATHOS.RUTGERS.EDU
ATHOS.RUTGERS.EDU has address 128.6.5.46
ATHOS.RUTGERS.EDU has address 128.6.4.4
ATHOS.RUTGERS.EDU mail is handled by ARAMIS.RUTGERS.EDU
```

The first line indicates that the name *sun4.rutgers.edu* is actually a nickname. The official host name is *ATHOS.RUTGERS.EDU*. The next two lines show the address(es). If a system has more than one network interface, there will be a separate address line for each interface. The last line indicates that *ATHOS.RUTGERS.EDU* does not receive its own mail. Mail for it is taken by *ARAMIS.RUTGERS.EDU*. There may be more than one such line, since some systems have more than one other system that will handle mail for them. Technically, every system that can receive mail is supposed to have an entry of this kind. If the system receives its own mail, there should be an entry that mentions the system itself, for example ‘‘XXX mail is handled by XXX’’. However many systems that receive their own mail do not bother

to mention that fact. If a system has a ‘‘mail is handled by’’ entry, but no address, this indicates that it is not really part of the Internet, but a system that is on the network will forward mail to it. Systems on Usenet, Bitnet, and a number of other networks have entries of this kind.

Options

There are a number of options that can be used before the host name. Most of these options are meaningful only to the staff who have to maintain the domain database.

The option **–w** causes host to wait forever for a response. Normally it will time out after around a minute.

The option **–v** causes printout to be in a ‘‘verbose’’ format. This is the official domain master file format, which is documented in the manual page for *named* under UNIX. Without **–v**, ‘‘a’’, ‘‘mx’’, and ‘‘cname’’ records are written out as ‘‘has address’’, ‘‘mail is handled by’’, and ‘‘is a nickname for’’, and TTL and class fields are not shown.

The option **–r** causes recursion to be turned off in the request. This means that the name-server will return only data it has in its own database. It will not ask other servers for more information.

The option **–d** turns on debugging. Network transactions are shown in detail.

The option **–t** allows specification of a particular type of information to be looked up. The arguments are defined in the man page for *named*. Currently supported types are a, ns, md, mf, cname, soa, mb, mg, mr, null, wks, ptr, hinfo, minfo, mx, uinfo, uid, gid, unspec, and the wildcard, which may be written as either ‘‘any’’ or ‘‘*’’. Types must be given in lower case. Note that the default is to look first for ‘‘a’’, and then ‘‘mx’’, except that if the verbose option is turned on, the default is only ‘‘a’’.

The option **–a** (for ‘‘all’’) is equivalent to **–v –t any**.

The option **–l** causes a listing of a complete domain. For example,

```
host -l rutgers.edu
```

will give a listing of all hosts in the *rutgers.edu* domain. The **–t** option is used to filter what information is presented, as you would expect. The default is address information, which also include PTR and NS records. The command

```
host -l -v -t any rutgers.edu
```

will give a complete download of the zone data for rutgers.edu, in the official master file format. (However the SOA record is listed twice, for arcane reasons.) NOTE: **–l** is implemented by doing a complete zone transfer and then filtering out the requested information. This command should be used only if it is absolutely necessary.

Customizing Host Name Lookup

In general, if the name supplied by the user does not have any dots in it, a default domain is appended to the end. This domain can be defined in */etc/resolv.conf*, but is normally derived by taking the local hostname after its first dot. The user can override this, and specify a

different default domain, using the environment variable `LOCALDOMAIN`. In addition, the user can supply his own abbreviations for host names. They should be in a file consisting of one line per abbreviation. Each line contains an abbreviation, a space, and then the full host name. This file must be pointed to by an environment variable `HOSTALIASES`, which is the name of the file.

Warnings

Unexpected effects can happen when a name is typed that is not part of the local domain. Always keep in mind the fact that the local domain name is tacked onto the end of every name, unless it ends in a dot. Only if this fails is the name used unchanged.

The **–l** option only tries the first name-server listed for the domain that you have requested. If this server is dead, you may need to specify a server manually. For example, to get a listing of *foo.edu*, one could try

```
host -t ns foo.edu
```

to get a list of all the name-servers for *foo.edu*, and then try

```
host -l foo.edu xxx
```

for all xxx on the list of name-servers, until one is found that works.

Name

hostname – print the name of the current system

Synopsis

```
hostname
```

Description

Hostname prints the name of the system as defined in the file */super/admin/Hostname*. Since Amoeba appears as a single host to the user there is only one host name for the entire collection of processors comprising a system.

Files

/super/admin/Hostname contains the name of the system.

Name

jove – emacs-like text editor

Synopsis

```
jove [-d directory] [-w] [-t tag] [+[n] file] [-p file] [files]
```

Description

Jove is Jonathan's Own Version of Emacs. It is based on the original *Emacs* editor written at MIT by Richard Stallman. Although *jove* is meant to be compatible with *Emacs*, there are some major differences between the two editors and you should not rely on their behaving identically.

Jove works on any reasonable display terminal that is described in the *termcap* file (see *termcap*(U) for more details). When you start up *jove*, it checks to see whether you have your `TERM` environment variable set. On most systems that will automatically be set up for you, but if it is not *jove* will ask you what kind of terminal you are using. To avoid having to type this every time you run *jove* you can set your `TERM` environment variable yourself. How you do this depends on which shell you are running. In the Bourne Shell and the Korn Shell, you type

```
$ TERM=type; export TERM
```

where *type* is the name of the kind of terminal you are using (e.g., vt100). If neither of these works get somebody to help you.

If you run *jove* with no arguments you will be placed in an empty buffer, called *Main.* Otherwise, any arguments you supply are considered file names and each is given its own buffer. Only the first file is actually read in — reading other files is deferred until you actually try to use the buffers they are attached to. This is for efficiency's sake: most of the time, when you run *jove* on a big list of files, you end up editing only a few of them.

The names of all of the files specified on the command line are saved in a buffer, called **minibuf**. The mini-buffer is a special *jove* buffer that is used when *jove* is prompting for some input to many commands (for example, when *jove* is prompting for a file name). When you are being prompted for a file name, you can type C-N (that is Control-N) and C-P to cycle through the list of files that were specified on the command line. The file name will be inserted where you are typing and then you can edit it as if you typed it in yourself.

Options

Jove recognizes the following options:

–d *directory*
: The *directory* argument is taken to be the name of the current directory.

+[*n*] *file*
: Reads the file, designated by the following argument, and positions the current point at the *n*'th line instead of the (default) 1st line. This can be specified more than once but it does not make sense to use it twice on the same file; in that case the second one wins. If no numeric argument is given after the +, the point is positioned at the end of the file.

−p *file*
Parses the error messages in the designated *file*. The error messages are assumed to be in a format similar to the C compiler, or *grep*(U) output.

−t *tag*
Runs the `find-tag` command on the string of characters immediately following the **−t** if there is one (as in ‘‘**−t**Tagname’’), or on the following argument (as in ‘‘**−t** Tagname’’) otherwise (see *ctags*(U)).

−w Divides the window in two. When this happens, either the same file is displayed in both windows, or the second file in the list is read in and displayed in its window.

Getting Help

Once in *jove*, there are several commands available to get help. To execute any *jove* command, you type ‘‘<ESC> X command-name’’ followed by <Return>. To get a list of all the *jove* commands you type ‘‘<ESC> X’’ followed by ‘‘?’’. The `describe-bindings` command can be used to get a list containing each key, and its associated command (that is, the command that gets executed when you type that key). If you want to save the list of bindings, you can set the jove variable `send-typeout-to-buffer` to ‘‘on’’ (using the `set` command), and then execute the `describe-bindings` command. This will create a buffer and put in it the bindings list it normally would have printed on the screen. Then you can save that buffer to a file and print it to use as a quick reference card. (See *Variables* below.)

Once you know the name of a command, you can find out what it does with the `describe-command` command, which you can invoke quickly by typing ‘‘ESC ?’’. The `apropos` command will give you a list of all the command with a specific string in their names. For example, if you want to know the names of all the commands that are concerned with windows, you can run `apropos` with the keyword `window.`

If you are not familiar with the *Emacs* command set, it would be worth your while to run *teachjove*. To do that, just type **teachjove** to the shell and you will be placed in *jove* in a file which contains directions. This is highly recommended for beginners; it may save a lot of time and headaches.

Key Bindings and Variables

You can alter the key bindings in *jove* to fit your personal tastes. That is, you can change what a key does every time you strike it. For example, by default the C-N key is bound to the command `next-line` and so when you type it you move down a line. If you want to change a binding or add a new one, you use the `bind-to-key` command. The syntax is ‘‘bind-to-key <command> key’’.

You can also change the way *jove* behaves in little ways by changing the value of some variables with the `set` command. The syntax is ‘‘set <variable> value’’, where value is a number or a string, or ‘‘on’’ or ‘‘off’’, depending on the context. For example, if you want *jove* to make backup files, you set the `make-backup-files` variable to `on`. To see the value of a variable, use the ‘‘print <variable>’’ command.

Initialization

Jove automatically reads commands from an initialization file called *.joverc* in your HOME directory, In this file you can place commands that you would normally type in *jove*. If you like to rearrange the key bindings and set some variables every time you get into *jove*, you should put them in your initialization file. Here are a few examples:

```
set match-regular-expressions on
auto-execute-command auto-fill /tmp/Re\|.*drft
bind-to-key i-search-forward  ^\
bind-to-key i-search-reverse  ^R
bind-to-key find-tag-at-point ^[^T
bind-to-key scroll-down       ^C
bind-to-key grow-window       ^Xg
bind-to-key shrink-window     ^Xs
```

(Note that the Control Characters can be either two character sequences (e.g., ^ and C together as ^C) or the actual control character. If you want to use an ^ by itself you must BackSlash it (e.g., ‘‘ `bind-to-key grow-window ^X\^` ’’ binds `grow-window` to `^X^`).

Some minor details

You may have to type C-\ instead of C-S in some cases. For example, the way to search for a string is documented as being ‘‘C-S’’ but in reality you should type ‘‘C-\’’. This is because C-S is the XOFF character (what gets sent when you type the NO-SCROLL key), and clearly that will not work. The XON character is ‘‘C-Q’’ (what gets sent when you type NO-SCROLL again) which is documented as the way to do a quoted-insert. The alternate key for this is ‘‘C-^’’ (typed as ‘‘C-`’’ on vt100s and its look-alikes). If you want to enable C-S and C-Q and you know what you are doing, you can put the line:

```
set allow-^S-and-^Q on
```

in your initialization file.

If your terminal has a metakey, *jove* will use it if you turn on the `meta-key` variable. *Jove* will automatically turn on `meta-key` if the METAKEY environment variable exists. This is useful for if you have different terminals (e.g., one at home and one at work) and one has a metakey and the other does not.

Files

/profile/module/jove/.joverc	system wide initialization file
$HOME*/.joverc*	personal initialization file
/tmp	where temporary files are stored
/profile/module/jove/teach-jove	the document used by the interactive tutorial *teachjove*

Diagnostics

Jove diagnostics are meant to be self-explanatory, but you are advised to seek help whenever you are confused. You can easily lose a lot of work if you do not know *exactly* what you are doing.

Warnings

Lines cannot be more than 1024 characters long.

Searches cannot cross line boundaries.

Name

kill – send a signal to a process or process group

Synopsis

```
kill [-sig] pid
kill -sig -pgrp
```

Description

Kill is used to send a signal to the process specified by the process identifier *pid*. By default signal 15 (`SIGTERM`) is sent. An alternative signal can be specified with the **–sig** option. Pid 0 means all the processes in the sender's process group are signaled.

If the **–sig** argument is specified and the second argument is prefixed with a minus then it is interpreted as a process group (*pgrp*) and the signal is sent to all processes in that process group.

The *pid* and *pgrp* for a process can be obtained using *aps*(U).

Examples

```
kill 35
```

Send signal 15 to process 35.

```
kill -9 40
```

Send signal 9 to process 40.

```
kill -2 0
```

Send signal 2 to the each process in the current shell's process group.

```
kill -3 -5
```

Send signal 3 to each process in process group 5.

See Also

aps(U), stun(U).

Name

ksh – the Korn Shell (Public Domain)

Synopsis

```
ksh [-st] [-c command] [file [argument ...]]
```

Description

The Korn Shell is an alternative command shell. It is Bourne Shell compatible (so shell scripts are portable), and its main feature is the ability to edit the command line (including command and filename completion and history editing).

Characteristics common to the Bourne and Korn Shell are only discussed briefly. Specific features of the Korn shell are described in more detail. Only a subset of the Korn shell features is currently implemented.

Command syntax

The '#' character begins a one-line comment, unless the '#' occurs inside a word. The tokens ';', '|', '&', ';;', '||', '&&', '(', and ')' stand by themselves. A *word* is a sequence of any other non-white space characters, which may also contain quoted strings (quote characters are '´', '"', '`', or a matching '${ }' or '$( )' pair). A *name* is an unquoted word made up of letters, digits, or '_'. Any number of white space characters (space and tab) may separate words and tokens.

In the following syntax, '{ ... }?' indicates an optional thing, '{ ... }*' indicates zero or more repetitions, '{ ... | ... }' indicates alternatives. Finally, '*char*' is used when character *char* is meant literally. Keywords are written in bold format.

statement:

'(' *list* ')'
'{' *list* ';' '}'
for *name* { **in** { *word* }* }? **do** *list* ';' **done**
{ **while** | **until** } *list* ';' **do** *list* ';' **done**
if *list* ';' **then** *list* ';' { **elif** *list* ';' **then** *list* ';' }* { **else** *list* ';' }?**fi**
case *name* **in** { *word* { '|' *word* }* ')' *list* ';;' }* **esac**
function *name* '{' *list* ';' '}'
name '()' '{' *list* ';' '}'
time *pipe*
Redirection may occur at the beginning or end of a statement.

command:

{ *name*'='*word* }* { *word* }*
This creates a temporary assignment to variable *name*, during the executing the second part. Redirection may occur anywhere in a command.

list:

cond
cond ';' *list*
cond '&' *list*

cond:

pipe
pipe '&&' *cond*
pipe '||' *cond*

pipe:

statement { '|' *statement* }*

Alias expansion

Alias expansion occurs when the first word of a statement is a defined alias, except when that alias is already being expanded. It also occurs after the expansion of an alias whose definition ends with a space.

Shell variables

The following standard special variables exist: '!', '#', '$', '-', '?'.

_	In interactive use this parameter is set to the last word of the previous command. When a command is executed this parameter is set to the full path of the command and placed in the environment for the command. See also MAILPATH.
CDPATH	The search path for the *cd* command.
ENV	If this variable is set at start-up (after any profile files are executed), the expanded value is used as shell start-up file. It typically contains function and alias definitions.
FCEDIT	The editor used by the *fc* command.
IFS	*Internal field separator*, used during substitution and the *read* command.
HOME	The default directory for the *cd* command.
MAIL	If set, the user will be informed of the arrival of mail in the named file. This variable is ignored if the MAILPATH variable is set.
MAILCHECK	How often, in seconds, the shell will check for mail in the file(s) specified by MAIL or MAILPATH. If 0, the shell checks before each prompt. The default is 600 seconds.
MAILPATH	A list of files to be checked for mail. The list is colon separated, and each file may be followed by a **?** and a message to be printed if new mail has arrived. Command and parameter substitution is performed on the message, and the parameter **$**_ is set to the name of the file. The default message is ''you have mail in $_''.
PATH	The search path for executable commands and **.**'d files.
PPID	The process number of the parent of the shell.
PS1	
PS2	PS1 is the primary prompt for interactive shells. Dollar substitution is performed, and variable '!' is replaced with the command number (see *fc*). The prompt $PS2 is printed when a statement is not yet complete, after having typed return.

PWD

OLDPWD

The current and previous working directories.

RANDOM A random integer. The random number generator may be seeded by assigning an integer value to this variable.

SECONDS The number of seconds since the shell timer was started or reset. Assigning an integer value to this variable resets the timer.

Substitution

In addition to the System V.2 substitutions, the following are available.

$(command) Like `command`, but no escapes are recognized.

$(<file) Equivalent to $(cat file), but without forking.

${#var} The length of the string value of *var*, or the number of arguments if *var* is ***** or **@**.

${var#pattern}

${var##pattern}

If *pattern* matches the beginning of the value of *var*, the matched text is deleted from the result of substitution. A single **#** results in the shortest match, two **#**'s results in the longest match.

${var%pattern}

${var%%pattern}

Like **#** substitution, but deleting from the end of the value.

Expressions

Expressions can be used with the *let* command, as numeric arguments to the *test* command, and as the value of an assignment to an integer variable.

Expression may contain alpha-numeric variable identifiers and integer constants and may be combined with the following operators: == != <= < > >= + - * / % ! ( )

Command execution

After evaluation of keyword assignments and arguments, the type of command is determined. A command may execute a shell function, a shell built-in, or an executable file.

Any keyword assignments are then performed according to the type of command. In function calls assignments are local to the function. Assignments in built-in commands marked with a † persist, otherwise they are temporary. Assignments in executable commands are exported to the sub-process executing the command.

There are several built-in commands.

: Only expansion and assignment are performed. This is the default if a command has no arguments.

. *file* Execute the commands in *file* without forking. The file is searched in the directories of $PATH. Passing arguments is not implemented.

alias [*name=value* ...]

Without arguments, *alias* lists all aliases and their values. For any name without a value, its value is listed. Any name with a value defines an alias, see "Alias Expansion" above. Korn's tracked aliases are not implemented, but System V command hashing is (see "hash").

alias -d [*name=value ...]*

Directory aliases for tilde expansion, e.g.

```
alias -d fac=/usr/local/usr/facilities
cd ~fac/bin
```

break [*levels*]

builtin *command arg* ...

Command is executed as a built-in command.

cd [*path*] Set the working directory to *path*. If the variable CDPATH is set, it lists the search path for the directory containing *path*. A null path means the current directory. If *path* is missing, the home directory ($HOME) is used. If *path* is **–**, the previous working directory is used. If *path* is **..**, the shell changes directory to the parent directory, as determined from the value of PWD. The PWD and OLDPWD variables are reset. The System V two argument form is not implemented.

cd *old new* The string *new* is substituted for *old* in the current directory, and the shell attempts to change to the new directory.

continue [*levels*]

echo ... *Echo* is replaced with the alias echo='print' in the Korn shell.

eval *command* ...

exec *command arg* ...

The executable command is executed without forking. If no arguments are given, any I/ O redirection is permanent.

exit [*status*]

fc [**–e** *editor*] [**–lnr**] [*first* [*last*]]

A simple subset of Korn's ''fix command''. *First* and *last* select commands. Commands can be selected by history number, or a string specifying the most recent command starting with that string. The **–l** option lists the command on stdout, and **–n** inhibits the default command numbers. The **–r** option reverses the order of the list. Without **–l**, the selected commands can be edited by the editor specified with the **–e** option, or if no **–e** is specified, the **$FCEDIT** editor, then executed by the shell.

fc **–e –** [**–g**] [*old=new*] [*command*]

Re-execute the selected command (the previous command by default) after performing the optional substitution of *old* with *new*. If **–g** is specified, all occurrences of *old* are replaced with *new*. This command is usually accessed with the predefined alias r=''fc –e –''.

getopts *optstring name* [*arg* ...]

Used for option handling in shell procedures. This command is described in a

separate section.

hash [**−r**] [*name* ...]

Without arguments, any hashed executable command path names are listed. The **−r** flag causes all hashed commands to be removed. Each *name* is searched as if it were a command name and added to the hash table if it is an executable command.

jobs Display information about the controlled jobs. The job number is given preceded by a percent sign, followed by a plus sign if it is the ‘‘current job’’, or by a minus sign if it is the ‘‘previous job’’, then the process group number for the job, then the command.

kill [**−***signal*] *process* ...

Send a signal (TERM by default) to the named process. The signal may be specified as a number or a mnemonic from <signal.h> with the SIG prefix removed.

let [*expression* ...]

Each expression is evaluated, see "Expressions" above. A zero status is returned if the last expression evaluates to a non-zero value, otherwise a non-zero status is returned. Since may expressions need to be quoted, *((expr))* is syntactic sugar for *let "expr"*.

print [**−nre**] [**−u***n*] [*argument* ...]

Print prints its arguments on the standard output, separated by spaces, and terminated with a newline. The **−n** option eliminates the newline.

By default, certain C escapes are translated. These include \b, \f, \n, \r, \t, \v, and *ooo* (*o* is an octal digit). \c is equivalent to the **−n** option. This expansion may be inhibited with the **−r** option, and may be re-enabled with the addition of the **−e** option.

The **−u** option can be used to let the results be written on the file descriptor specified (e.g., ‘‘`print -u2`’’ writes on stderr).

read [**−r**] [**−u***n*] *name* ...

Read the value of variables specified from standard input. The **−u** option can be used to specify a different file descriptor. The first variable name may be of the form *name***?***prompt*.

readonly [*name* ...]

Make the variables specified read-only, i.e., future assignments to them will be refused.

return [*status*]

Used to return from a user defined function, with *status* specifying the return value (default 0).

set [±[*a-z*]]

Set (**−**) or clear (**+**) a shell option:

−a allexport all new variable are created with export attribute
−e errexit exit on non-zero status
−k keyword variable assignments are recognized anywhere in command

−n noexec	compile input but do not execute (ignored if interactive)
−f noglob	do not expand filenames
−u nounset	dollar expansion of unset variables is an error
−v verbose	echo shell commands on stdout when compiling
−h trackall	add command path names to hash table
−x xtrace	echo simple commands while executing

set [±**o** *keyword*] ...

Set (**−**) or clear (+) shell option *keyword*:

emacs	emacs-like line editing
ignoreeof	shell will not exit of EOF, must use *exit*
vi	VI-like line editing

set [**−−**] *arg* ...

Set shell arguments.

shift [*number*]

test *expression*

The *test* command has been integrated into the shell, for efficiency. *Test* evaluates the *expression* and returns zero status if true, and non-zero status otherwise. It is normally used as the controlling command of the *if* and *while* statements. See test(U) for a complete description.

times

trap [*handler*] [*signal* ...]

typeset [±**irtx**] [*name*[=*value*] ...]

If no arguments are given, lists all variables and their attributes.

If options but no names are given, lists variables with specified attributes. Their values are listed also, unless ‘‘+’’ is used.

If names are given, set the attributes of the named variables. Variables may also be assigned a value. If used inside a function, the created variable are local to the function.

The attributes are as follows:

- -i The variable’s value is stored as an integer.
- -x The variable is exported to the environment.
- -r The variable is read-only cannot be reassigned a value.
- -f List functions instead of variable.

umask [*value*]

unalias *name* ...

The aliases for the given names are removed.

unset [**−f**] *name* ...

wait [*process-id*]

If *process-id* is given, wait for the specified process to finish. Otherwise wait for any process to finish.

whence [**−v**] name ...

For each name, the type of command is listed. The **−v** flag causes function and alias values to be listed.

Shell script option handling

The built-in command *getopts* can be used by shell procedures to parse positional parameters and to check for legal options. It supports all applicable rules of the command syntax standard. The syntax is ‘‘getopts *optstring name* [*arg* ...]’’

Optstring must contain the option letters the command using *getopts* will recognize; if a letter is followed by a colon, the option is expected to have an argument which should be separated from it by white space.

Each time it is invoked, *getopts* will place the next option in the shell variable *name* and the index of the next argument to be processed in the shell variable OPTIND. Whenever the shell or a shell procedure is invoked, OPTIND is initialized to 1.

When an option requires an option-argument, *getopts* places it in the shell variable OPTARG. If an illegal option is encountered, **?** will be placed in *name*. When the end of the options is encountered, *getopts* exits with a non-zero exit status. The special option may be used to delimit the end of the options. By default, *getopts* parses the positional parameters. If extra arguments ...) are given on the *getopts* command line, *getopts* will parse them instead.

The following fragment of a shell program shows how one might process the arguments for a command that can take the options **a** or **b**, as well as the option **o**, which requires an option-argument:

```
while getopts abo: c
do
        case $c in
        a|b)    FLAGS=$FLAGS$c;;
        o)      OARG=$OPTARG;;
        \?)     echo $USAGE 1>&2
                exit 2;;
        esac
done
shift OPTIND-1
```

This code will accept any of the following as equivalent:

```
cmd -a -b -o "xxx z yy" file
cmd -a -b -o "xxx z yy" -- file
cmd -ab -o "xxx z yy" file
cmd -ab -o "xxx z yy" -- file
```

Interactive Input Line Editing

When the *emacs* option is set, interactive input line editing is enabled. This mode is slightly different from the emacs mode in AT&T’s KornShell. In this mode various *editing commands* (typically bound to one or more control characters) cause immediate actions without waiting for a new-line. Several editing commands are bound to particular control characters when the shell is invoked; these bindings can be changed using the following commands:

bind The current bindings are listed.

bind *string=editing-command*

The specified editing command is bound to the given *string*, which should consist of a control character (which may be written using ‘‘caret notation’’ ^*x*), optionally preceded by one of the two prefix characters. Future input of the *string* will cause the editing command to be immediately invoked.

Note that although only two prefix characters (normal ESC and ^X) are supported, some multi-character sequences can be supported:

bind ’^[[’=prefix-2
bind ’^XA’=up-history
bind ’^XB’=down-history
bind ’^XC’=forward-char
bind ’^XC’=backward-char

will bind the arrow keys on an ANSI terminal. Of course some escape sequences will not work out quite that nicely.

bind -m *string=substitute*

The specified input *string* will afterwards be immediately replaced by the given *substitute* string, which may contain editing commands.

The following editing commands are available; first the command name is given followed by its default binding (if any) using caret notation (note that the ASCII ESC character is written as ^[ ), then the editing function performed is described. Note that editing command names are used only with the *bind* command. Furthermore, many editing commands are useful only on terminals with a visible cursor. The default bindings were chosen to resemble corresponding EMACS key bindings. The users tty characters (e.g. erase) are bound to reasonable substitutes.

abort ^G Useful as a response to a request for a *search-history* pattern in order to abort the search.

auto-insert Simply causes the character to appear as literal input. (Most ordinary characters are bound to this.)

backward-char ^B

Moves the cursor backward one character.

backward-word ^[b

Moves the cursor backward to the beginning of a word.

beginning-of-line ^A

Moves the cursor to the beginning of the input line (after the prompt string).

complete ^[^[Automatically completes as much as is unique of the hashed command name or the file name containing the cursor. If the entire remaining command or file name is unique a space is printed after its completion, unless it is a directory name in which case / is postpended. If there is no hashed command or file name with the current partial word as its prefix, a bell character is output (usually causing a ‘‘beep’’).

complete-command ^X^[

Automatically completes as much as is unique of the hashed command name having the partial word up to the cursor as its prefix, as in the *complete* command described above. Only command and function names

seen since the last *hash −r* command are available for completion; the *hash* command may be used to register additional names.

complete-file ^[^X
Automatically completes as much as is unique of the file name having the partial word up to the cursor as its prefix, as in the *complete* command described above.

copy-last-arg ^[_
The last word of the previous command is inserted at the cursor. Note I/O redirections do not count as words of the command.

delete-char-backward ERASE
Deletes the character before the cursor.

delete-char-forward ^D
Deletes the character after the cursor.

delete-word-backward ^[ERASE
Deletes characters before the cursor back to the beginning of a word.

delete-word-forward ^[d
Deletes characters after the cursor up to the end of a word.

down-history ^N
Scrolls the history buffer forward one line (later). Each input line originally starts just after the last entry in the history buffer, so *down-history* is not useful until either *search-history* or *up-history* has been performed.

end-of-line ^E Moves the cursor to the end of the input line.

eot ^_ Acts as an end-of-file; this is useful because edit-mode input disables normal terminal input canonicalization.

eot-or-delete ^D
Acts as eot if alone on a line; otherwise acts as delete-char-forward.

exchange-point-and-mark ^X ^X
Places the cursor where the mark is, and sets the mark to where the cursor was.

forward-char ^F
Moves the cursor forward one position.

forward-word ^[f
Moves the cursor forward to the end of a word.

kill-line KILL Deletes the entire input line.

kill-region ^W Deletes the input between the cursor and the mark.

kill-to-eol ^K Deletes the input from the cursor to the end of the line.

list ^[? Prints a sorted, columnar list of hashed command names or file names (if any) that can complete the partial word containing the cursor. Directory names have / postpended to them, and executable file names are followed by *.

list-command ^X ?
Prints a sorted, columnar list of hashed command names (if any) that can

complete the partial word containing the cursor.

list-file Prints a sorted, columnar list of file names (if any) that can complete the partial word containing the cursor. File type indicators are postpended as described under *list* above.

newline ^J *and* **^M**
Causes the current input line to be processed by the shell. (The current cursor position may be anywhere on the line.)

newline-and-next ^O
Causes the current input line to be processed by the shell, and the next line from history becomes the current line. This is only useful after an up-history or search-history.

no-op QUIT Does nothing.

prefix-1 ^[Introduces a 2-character command sequence.

prefix-2 ^X Introduces a 2-character command sequence.

quote ^^ The following character is taken literally rather than as an editing command.

redraw ^L Reprints the prompt string and the current input line.

search-character ^]
Search forward in the current line for the next keyboard character.

search-history ^R
Enter incremental search mode. The internal history list is searched backwards for commands matching the input. An initial ‘‘^’’ in the search string anchors the search. The escape key will leave search mode. Other commands will be executed after leaving search mode (unless of course they are prefixed by escape, in which case they will almost certainly do the wrong thing). Successive *search-history* commands continue searching backward to the next previous occurrence of the pattern. The history buffer retains only a finite number of lines; the oldest are discarded as necessary.

set-mark-command ^] <space>
Search forward in the current line for the next keyboard character.

stuff On systems supporting it, pushes the bound character back onto the terminal input where it may receive special processing by the terminal handler.

stuff-reset Acts like *stuff*, then aborts input the same as an interrupt.

transpose-chars ^T
Exchanges the two characters on either side of the cursor, or the two previous characters if the cursor is at end of line.

up-history ^P Scrolls the history buffer backward one line (earlier).

yank ^Y Inserts the most recently killed text string at the current cursor position.

yank-pop ^[y Immediately after a *yank*, replaces the inserted text string with the next previous killed text string.

Files

/home/.profile default startup file

Diagnostics

getopts prints an error message on the standard error output when it encounters an option letter not included in *optstring*.

Warnings

The 8th bit is stripped in emacs mode.

Options with option-arguments must not be grouped with other options, because this may not be supported anymore in a future release.

For the same reason, there should be white space after an option that takes an option-argument.

Changing the value of the shell variable OPTIND or parsing different sets of arguments may lead to unexpected results.

Quoting double-quote (") characters inside back-quote (`) inside double-quotes does not behave properly.

The emacs mode can ‘‘lose’’ an stty command done by the user.

Unsetting special variables may cause unexpected results.

Functions declared as having local scope really have global scope.

Here documents inside functions do not work correctly.

Exit on error (**set –e** or **set -o errexit**) does not work correctly.

Furthermore, there are several differences with the AT&T version:

The *select* statement is not implemented.

Variable arrays are not implemented.

Variable attributes other than integer are not implemented.

The ERR and EXIT traps are not implemented for functions.

Alias expansion is inhibited at the beginning of an alias definition in the AT&T version.

Korn evaluates expressions differently.

See Also

sh(U), test(U).

Name

ln – create a link to a file

Synopsis

```
ln file [name]
```

Description

A directory entry is created for name. The entry points to file. Henceforth, name and file can be used interchangeably. If name is not supplied, the last component of file is used as the link name.

Examples

```
ln file newname
```

Make newname a synonym for file.

```
ln /usr/games/chess
```

Create a link called *chess*.

See Also

get(U), put(U).

Name

look – look up words in dictionary

Synopsis

```
look [-f] prefix[/suffix] [dictionary]
```

Description

Look takes a *prefix* and/or *suffix* and searches */usr/lib/dictionary* or the specified dictionary for all words that have that prefix or suffix The words are printed.

Options

–f Fold upper case letters to lower case.

Examples

```
look ard
```

Print words starting with *ard*.

```
look /bing
```

Print words ending with *bing*.

```
look -f f/ar
```

Print words starting with *f*, ending with *ar*.

Name

ls – list the contents of a directory

Synopsis

```
ls [-1ACFRacdfgilrstu] name ...
```

Description

For each file argument, list it. For each directory argument, list its contents, unless **-d** is present. When no argument is present, the working directory is listed.

Options

–1 Make a single column listing.

–A All entries are listed, except . and ..

–C Multicolumn listing (default).

–F Put / after directory names.

–R Recursively list subdirectories.

–a All entries are listed, even file names starting with a ‘‘.’’.

–c Print the i-node last change time. This is same to the last modified time in Amoeba.

–d Do not list contents of directories.

–f List argument as unsorted directory.

–g Group id given instead of user id.

–i Print the object number of the directory entry. This is not unique since two objects of different types may have the same object number.

–l Long listing: mode, links, owner, size and time.

–r Reverse the sort order.

–s Give size in blocks (including indirect blocks).

–t Sort by time, latest first.

–u Use last usage time instead of modification time. This is same to the last modified time in Amoeba.

Warning

Ls uses the POSIX emulation which attempts to map Amoeba protection information to POSIX protection information. On the whole such mappings are difficult to get right. For example, objects that are neither directory or file are often described by *ls* as character special devices. To discover the true type and protection of an object use *dir*(U).

Examples

```
ls -l
```

List files in current directory.

```
ls -Rs /home/bin
```

List */home/bin* recursively with object sizes, where known.

See Also

dir(U).

Name

m2 – ACK Modula-2 compiler

Synopsis

```
m2 [options] file ...
```

Description

This document provides a short introduction to the use of the ACK Modula-2 compiler. It also tells where to find definition modules for ‘‘standard’’ modules.

File Names

Usually, a Modula-2 program consists of several definition and implementation modules, and one program module. Definition modules must reside in files with names having a ‘‘.def’’ suffix. Implementation modules and program modules must reside in files having a ‘‘.mod’’ suffix.

The name of the file in which a definition module is stored must be the same as the module-name, apart from the extension. For historic reasons, the compiler truncates to 10 characters the basename of the definition modules it tries to read. So, given an IMPORT declaration for a module called *LongModulName*, the compiler will try to open a file called *LongModulN.def*. This requirement does not hold for implementation or program modules, but is certainly recommended.

Calling the compiler

The easiest way to call the compiler is to let the *ack*(U) program do it. So, to compile a program module *prog.mod*, just call

`m2` –m*mach* prog.mod [objects of implementation modules]

where *mach* is one of the target machines of ACK. If there are no errors, this will produce a binary program *prog* that can be run on a machine of type *mach*. When no **–m** option is given, the architecture of the driver itself is assumed. The program *m2* is actually a link to the generic driver *ack*, which can be used to compile any type of source file supported by ACK (see *ack*(U) for a full list of options available).

To compile an implementation module, use the **–c** flag to produce a *.o* file. Definition modules can not be compiled; the compiler reads them when they are needed.

Definition Modules

‘‘Standard’’ definition modules can be found in the directory */profile/module/ack/lib/m2*. When the compiler needs a definition module, it is first searched for in the current directory, then in the directories given to it by the **–I** flag in the order given, and then in the directory mentioned above.

Options

The *ack*(U) program recognizes (among others) the following flags, that are passed to the Modula-2 compiler:

–I*dirname*
: append *dirname* to the list of directories where definition modules are looked for.

–I
: do not look in the directory */profile/module/ack/lib/m2*.

–M*n*
: set maximum identifier length to *n*. The minimum value of *n* is 14, because the keyword ‘‘IMPLEMENTATION’’ is that long.

–d*file*
: generate a list of *amake*(U) dependencies — consisting of a list of all included files — on the file specified.

–n
: do not generate EM register messages. The user-declared variables will not be stored into registers on the target machine.

–w[*classes*]
: suppress warning messages whose class is a member of *classes*. Currently, there are three classes: **O**, indicating old-fashioned use, **W**, indicating ‘‘ordinary’’ warnings, and **R**, indicating restricted Modula-2. If no *classes* are given, all warnings are suppressed. By default, warnings in class **O** and **W** are given.

–W*classes*
: allow for warning messages whose class is a member of *classes*.

–x
: make all procedure names global, so that a debugger understands them.

–Xs
: make INTEGER ranges symmetric, that is, MIN(INTEGER) = –MAX(INTEGER). This is useful for interpreters that use the ‘‘real’’ MIN(INTEGER) to indicate ‘‘undefined.’’

Examples

The directory *src/ack/examples/modula2* contains a Modula-2 version of an Amoeba client/server performance test. It shows how Modula-2 programs can call functions in the Amoeba libraries.

Diagnostics

All warning and error messages are written on standard error output.

Files

/profile/module/ack/bin/m2 – the compiler driver.

/profile/module/ack/lib/front/em_m2 – the Modula-2 front end.

See Also

ack(U), amake(U).

Name

m4 – macro processor

Synopsis

```
m4 [-D name = value] [-U name]
```

Description

M4 is a general-purpose macro processor. It has been used to implement programming languages, such as RATFOR.

Options

–D name = value
Define a symbol.

–U name
Undefine a symbol.

Example

```
m4 < m4test
```

Name

make – a program for maintaining medium-sized programs

Synopsis

```
make [-f file] [-inpqrst] ... [target]
```

Description

Make is a program that is normally used for developing medium-sized programs consisting of multiple files. It keeps track of which object files depend on which source and header files. When called, it tries to do the minimum amount of recompilation necessary to bring the target file up to date.

Options

–f Use *file* as the *makefile*.

–i Ignore status returned by commands. Normally *make* stops if a command that it has started returns an error status.

–n Report, but do not execute the commands necessary to remake the target(s).

–p Print macros and targets.

–q Question up-to-dateness of target. The exit status is 0 if the target is up to date and 1 otherwise.

–r Rule inhibit; do not use default rules.

–s Silent mode. Do not print commands before executing them.

–t Touch files (making them up to date) instead of making them.

The file dependencies are expected in *makefile* or *Makefile*, unless another file is specified with **–f**. *Make* has some default rules built in, for example, it knows how to make *.o* files from *.c* files. Here is a sample makefile.

```
d=/home/test                            # d is a macro
program: head.o tail.o                  # program depends on these
        cc -o program head.o tail.o      # tells how to make program
        echo Program done.               # announce completion

head.o: $d/def.h head.c                 # head.o depends on these
tail.o: $d/var.h tail.c                 # tail.o depends on these
```

A complete description of *make* would require too much space here. For more information, see Feldman (1979). Many books on UNIX also discuss *make*.

Warnings

Make uses a file's modification time to see if it is ''out of date.'' As the SOAP directory server currently does not provide a very precise file modification time, problems may arise when a file is to be recompiled, shortly after it has been changed. Amoeba's own configuration manager, *amake*(U), does not have this problem, since it records the actual

capabilities of file objects. Before using *make*, look at the manual page for *amake* to see if it is suitable. It is much more effective.

Examples

```
make kernel
```

Make *kernel* up to date.

```
make -n file
```

Print the commands that needs to be executed to bring *file* up to date.

See Also

amake(U).

Name

makecap – create a random capability

Synopsis

```
makecap [-r rights] [-o objectnumber] [-p putcapname] [getcapname]
```

Description

Makecap is used to create a random *get* capability. The server port and the check word in the capability are chosen at random, and it is extremely unlikely that the capability points to any existing object. The new capability can be stored in the directory server under the name *getcapname*. If the *getcapname* argument is not present or is ‘‘–’’, then it writes the capability on standard output.

Options

–o *objectnumber*
: With the **–o** option, one can specify the object number in the capability. By default, this is zero.

–p *putcapname*
: When the **–p** option is supplied, *makecap* also stores the corresponding *put* capability in the directory server, under the name specified.

–r *rights*
: With the **–r** option, one can specify the rights bits in the capabilities. By default, the capability has all rights bits set.

The default base for *objectnumber* and *rights* is decimal, but prefixing the number with a ‘‘0x’’ will change the base to hexadecimal. If invalid number specifications are given they will be interpreted as 0.

Diagnostics

If either the rights or the object number are out of range, *makecap* reports a failure in the function *prv_encode*. Other error messages should be self-explanatory.

Example

To create a random capability ‘‘foo’’, type:

```
makecap foo
```

See Also

prv_encode(L), uniqport(L).

Name

makepool – let the run server control a new pool directory

Synopsis

```
makepool [-s path] [-p path] pooldir
```

Description

Makepool asks the run server to manage the processor pool directory *pooldir*, by asking it to create a new run object corresponding to *pooldir*. The directory *pooldir* should contain subdirectories named *arch*, containing the capabilities for hosts having architecture *arch*.

Future operations on the capability of the new run object (for example, *exec_findhost* which asks for the ‘‘best’’ host to execute a given process descriptor) will let the run server choose among the hosts in the corresponding pool directory. See *run*(A) for an overview of the operations available.

If a pool directory managed by a run server is modified (i.e., when hosts or architecture directories are added or removed) this should be made known to the run server by issuing a *std_touch*(U) command on the corresponding run capability.

Options

–s path Can be used to specify a non default run server to be used. The default used is `DEF_RUNSERVER` from *ampolicy.h* (typically this will be */profile/cap/runsvr/default*).

–p path Specifies an alternative name under which to install the capability for the new run object. The default is ‘‘*pooldir*`/.run`’’, where *pooldir* is the pool directory given as argument to *makepool*.

To avoid confusion, it is best to store the run capability in the corresponding pool directory. Furthermore, it is advisable to let the last component of the run object’s name contain a dot. Programs scanning pool directories (e.g., *aps*) avoid doing (illegal) operations on objects with such a name.

Diagnostics

These are all self explanatory. Possible causes for failure are inability or refusal of the run server to create a new object, or an error while trying to install the new run capability (to avoid erroneously throwing away a still valid capability, *makepool* only tries to append it to the publishing directory).

Example

The command

```
makepool /super/pool
```

creates a new run object for the public pool. The capability is stored in */super/pool/.run*. Requests to this run object will normally be issued to the run object */profile/pool/.run*, which is the same, but lacks a few administrative rights.

See Also

exec_findhost(L), run(A), std_touch(L).

Name

makeproto – modifies the output of mkproto to be more useful

Synopsis

```
makeproto [-n] [-s] [-p] [file] ...
```

Description

Makeproto is a shell script which calls *mkproto* to generate STD C prototypes for a C source file and then modifies the output of *mkproto* by deleting lines containing the words PRIVATE or LOCAL and converting the word PUBLIC to EXTERN. The result is generated on standard output. If no file name is specified in the command line it reads from standard input. The options are those of *mkproto*.

Example

```
makeproto foo.c > foo.pro
```

will generate STD C function prototypes for the routines in *foo.c* and writes them to the file *foo.pro*.

See Also

mkproto(U).

Name

mkd – make a new directory

Synopsis

```
mkd [-c column] pathname ...
```

Description

Mkd uses *sp_mkdir*(L) to create new directories with the given *pathnames*.

Options

–c column By default, *mkd* creates directories with three columns: *owner*, *group*, and *other*. A different directory structure can be forced by specifying the column names with the **–c** option explicitly, one for each column.

Examples

```
mkd /home/newdir
```

will create a new directory named *newdir*, under the */home* directory, which must exist before this command is invoked.

The command

```
mkd -c myself -c group /home/mydir
```

will create a directory named *mydir* having two columns, named *myself* and *group*, respectively.

See Also

chm(U), dir(U), get(U), del(U), put(U).

Name

mkdir – make a directory

Synopsis

```
mkdir directory ...
```

Description

The specified directory or directories are created. The defaults are used for directory protection. In particular, an attempt is made to interpret the umask, if present, but this is not reliable.

Examples

```
mkdir dir
```

Create *dir* in the current directory.

```
mkdir /home/dir
```

Create the directory *dir* in the directory */home*.

See Also

mkd(U).

Name

mkfifo – make a named pipe

Synopsis

```
mkfifo [-m mode] fifo ...
```

Description

Mkfifo It creates the specified fifos with the names specified. It does this by communicating with the *fifosvr*(A) server which implements fifos.

Mkfifo is a POSIX 1003.2 compliant utility.

Options

–m *mode*

This option causes the directory permissions of the new fifo(s) to be set to *mode*. The mode is specified symbolically.

Examples

```
mkfifo /tmp/foo
```

This creates a fifo called */tmp/foo*.

```
mkfifo -m a+w systatus
```

This creates a fifo called *systatus* and makes it writable for all users.

See Also

fifosvr(A).

Name

mkproto – make STD C function prototypes for a C source file

Synopsis

```
mkproto [-n] [-s] [-p] [file] ...
```

Description

Mkproto takes as input one or more C source code files, and produces as output (on the standard output stream) a list of function prototypes (a la STD C) for the external functions defined in the given source files. This output, redirected to a file, is suitable for #including in a C source file.

The function definitions in the original source may be either ‘‘old-style’’ (in which case appropriate prototypes are generated for the functions) or ‘‘new-style’’ (in which the definition includes a prototype already).

Options

–n causes the line number where each function was defined to be prepended to the prototype declaration as a comment.

–s causes prototypes to be generated for functions declared ‘‘static’’ as well as extern functions.

–p causes the prototypes emitted to be only readable by STD compilers. Normally, the prototypes are ‘‘macro-ized’’ so that compilers with __STDC__ not defined do not see them.

If files are specified on the command line, then a comment specifying the file of origin is emitted before the prototypes constructed from that file. If no files are given, then no comments are emitted and the C source code is taken from the standard input stream.

Warnings

Mkproto is easily confused by complicated declarations, such as

```
int (*signal())() { ...
```

or

```
struct foo { int x, y; } foofunc() { ...
```

Float types are not properly promoted in old style definitions, that is,

```
int test(f) float f; { ...
```

should (because of the default type conversion rules) have prototype

```
int test(double f);
```

rather than the incorrect

```
int test(float f);
```

generated by *mkproto*.

Some programs may need to be run through the preprocessor before being run through *mkproto*. The **–n** option is unlikely to work as desired on the output of a preprocessor.

Typedef'd types are not correctly promoted. For example, for

```
typedef schar char; int foo(x) schar x;...
```

mkproto incorrectly generates the prototype

```
int foo(schar x)
```

rather than the (correct)

```
int foo(int x)
```

Functions named ‘‘inline’’ with no explicit type qualifiers are not recognized.

Note

There is no warranty for this program (as noted above, it is guaranteed to break sometimes anyway).

Example

```
mkproto fred.c > fred.pro
```

will produce the file *fred.pro* which contains the STD C prototypes for all the externally visible functions defined in *fred.c*.

See Also

cc(U), makeproto(U).

Name

monitor – monitor a running server

Synopsis

```
monitor name [ subcommand [arg ... ] ]
```

Description

Monitor is used to get the monitor data being generated by a server. Monitoring data consists of a string which identifies an event, followed by a count of the number of times that event has occurred. (See *monitor*(H) for more details.) *Monitor* without a *subcommand* argument gets the capability *name* from the directory service and gets the monitoring data from the server whose port is in the capability. It then prints out all events that occurred in the server together with the counts of the number of times that they have occurred.

It is possible to switch the server's monitor into a mode where it will keep events in a circular buffer in the order in which they occurred. This requires more work by the server. The subcommand *circbuf on* tells the server to start this mode of operation. Thereafter the subcommand *circbuf* without arguments will print a list of events, since the last *circbuf* monitor request. As one would expect *circbuf off* turns this collecting off. Intervening calls to the monitor without the *circbuf* subcommand will return the normal statistics.

The subcommand *reset* will reset all counters to zero in the monitored server.

No meaningful results will be obtained if the server did not do any monitoring.

Diagnostics

Error messages arise because either the server is not running or the server is not doing monitoring. In the latter case the following error message will be printed:

```
monitor: Couldn't get monitor data: invalid capability
```

Example

```
monitor ''
```

will give the monitor data for the directory server, since `''` is the current directory and the server port in your home directory capability is that of the directory server.

```
monitor /dev/session circbuf on
```

will start circular buffer monitoring on the session server. Subsequent

```
monitor /dev/session circbuf
```

commands will print monitor data for the session server events that happened since the previous execution of the command.

```
monitor /dev/session reset
```

will set all the monitor counters to zero in the session server.

Name

more – a pager

Synopsis

```
more [-cdflpsu] [-linenum] [+linenum | +/pattern] file ...
```

Description

More is a pager that allows one to examine files. This program was originally produced at the University of California, Berkeley. It has been slightly modified for Amoeba. When *more* starts up, it displays a screen full of information from the first file in its list, and then pauses for one of the following commands.

In this description, ‘‘#’’ represents an integer telling how many of something.

<space>	Display next page
<return>	Display next line
CTRL-B	Go backward half a screen full
CTRL-D	Go forward half a screen full
CTRL-L	Redisplay the screen
#<space>	Go forward # lines
=	Print current line number
.	Repeat previous command
’	(single quote) Go back to start of last search
!	Escape to a shell
#/<expr>	Go to #-th occurrence of <expr>
:f	Display current file name and line number
#:n	Skip forward # files
#:p	Skip backward # files
b	Go backward half a screen full
d	Go forward half a screen full
#f	Skip # screen fulls
h	Display /usr/lib/more.help
#n	Go to #-th occurrence of last <expr>
q	Quit more
Q	Quit more
#s	Skip # lines
v	Try to execute */usr/bin/vi*
#z	Go forward # lines and set screen size to #

Options

–d Display prompt message at each pause.

–f Do not fold lines.

–l Do not treat CTRL-L as form feed.

–p Page mode. Do not scroll.

–s Suppress multiple blank lines.

–u Use escape sequences for underlining.

–*linenum*
: Display *linenum* lines of the file per page.

+*linenum*
: Display from line *linenum* in the first file.

+/*pattern*
: Search for the specified *pattern* in the first file and begin displaying from the first line containing *pattern*.

Environment Variables

For the benefit of users who always want to use certain flags when calling more, the string environment variable `MORE` can be set to the desired default flags, for example, `MORE`=‘‘–p’’.

Examples

```
more file
```

Display file on the screen.

```
more -p file1 file2
```

Display two files in page mode.

```
more -10 file
```

Use a 10 line window

```
more +/begin file
```

Hunt for the string ‘‘begin’’ in *file*.

See Also

yap(U).

Name

mv – move or rename an object

Synopsis

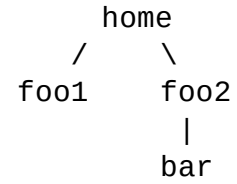
```
mv obj1 obj2
mv obj ... directory
```

Description

Mv moves one or more directory entries from one directory in the name server to another. Note that no copying of objects takes place. Only directory entries are moved.

Warnings

It is possible to ‘‘lose’’ a directory using *mv*.



For example, if in the directory */home* there is a file *foo1,* and a directory *foo2*, and if *foo2* has a subdirectory *bar*, then the command sequence

```
cd /home/foo2/bar
mv /home/foo* .
```

will also move the directory *foo2* to the directory *bar*, resulting in a directory graph cut off from */home*. Unless there is another link to *foo2* or *bar*, this orphan directory graph will be inaccessible and eventually be garbage collected.

Examples

```
mv oldname newname
```

Move the object *oldname* to *newname*.

```
mv file1 file2 /home
```

Move the two objects *file1* and *file2* to the directory */home*.

See Also

cp(U).

Name

notify – send a message to logged in users

Synopsis

```
notify group message
notify -u user message
```

Description

Notify is used to send a *message* to one or more logged-in users. The string

`Writing to` *user*`/dev/console`

is printed for each user that received the message. The *message* must be a single argument, so it should be between quotes if it contains white-space or characters special to the shell.

If sending to a group, the name of the group can either be an absolute path name or a name relative to */profile/group*. The *group* must be a directory containing the capabilities for the home directories of the group members. All members of the group who are currently logged in will receive the message unless they have restricted access to their */dev/console* entry.

If sending to a single user then the **–u** option should be used. The subsequent argument is then interpreted as the name of a user. If the *user* name is not an absolute path name it is interpreted as a name relative to */profile/group/users*.

Diagnostics

`cannot find your user name`
: The current user's login directory was not found in */profile/group/users*.

user`/dev/console not logged in`
: There was either no */dev/console* entry for *user* or it was not responding. This message normally only appears when the **–u** option was used and the user was not logged in.

Files

/dev/console– *notify* writes to this device to print its message.

/profile/group– this directory contains the groups to which messages can be sent.

Warnings

It is possible for other people to write on your session server's console window. The default is all the people in the groups you are in. To restrict access use *chm*(U) to restrict access to */dev/console*. This is best done in your login profile since the */dev/console* entry is remade with default permissions whenever you start a new *session*(U).

Examples

```
notify staff "the coffee is ready"
```

will send the message ‘‘the coffee is ready’’ to the group *staff*.

```
notify -u sergio "please come to my office"
```

will send the message ‘‘please come to my office’’ to the user with the login name *sergio*.

```
notify /super/users "system going down in 10 minutes"
```

This message can only be sent by a super-user and sends the message ‘‘system going down in 10 minutes’’ to all logged on users.

See Also

sess_setout(U), session(U).

Name

nm – print the symbol table of Amoeba executables

Synopsis

```
nm [-dgnopru] [file] ...
```

Description

Nm prints the symbol table of the Amoeba executables given as argument. When no argument is given, the symbol table of file *a.out* will be given. By default, all symbols are printed in alphabetic order, and each will be preceded by its address (in hexadecimal notation) and a letter telling the type of the symbol.

The type letters have the following meaning:

A absolute value

B symbol in bss segment

D symbol in data segment

T symbol in text segment

U undefined symbol

When the letter is printed in lower case, it means that the symbol is local rather than global.

Options

–d print addresses in decimal.

–g print only global symbols.

–n sort numerically by address rather than alphabetically by name.

–o prepend the file name to each line rather than only once.

–p print symbols in symbol table order.

–r sort in reverse order.

–u print only undefined symbols.

Example

```
nm /bin/nm/pd.i80386
```

Prints the symbol table of *nm* itself. A small subset of the output could be:

```
00017bd8 B _n_flag
00002efa T _name_lookup
00009392 t _newseg
```

Name

od – octal dump

Synopsis

```
od [-bcdhox] [file] [ [+] offset [.][b] ]
```

Description

Od dumps a file in one or more formats. If the *file* parameter is missing, *stdin* is dumped. The *offset* argument tells *od* to skip a certain number of bytes or blocks before starting. The offset is in octal bytes, unless it is followed by a ‘‘.’’ for decimal or b for blocks or both.

Options

–b Dump bytes in octal.

–c Dump bytes as ASCII characters.

–d Dump words in decimal.

–h Print addresses in hex (default is octal).

–o Dump words in octal (default).

–x Dump words in hex.

Examples

```
od -ox file
```

Dump file in octal and hex.

```
od -d file +1000
```

Dump file starting at byte 01000.

```
od -c file +10.b
```

Dumps a file starting at block 10.

Name

pc – ACK Pascal compiler

Synopsis

```
pc  [options] files
```

Description

Pc is the Pascal front end of the Amsterdam Compiler Kit (ACK). The compiler complies as much as possible with the ISO standard proposal. The language features as processed by this compiler are described in the Pascal reference manual in the Programming Guide. Normally the compiler is called by means of the user interface program *pc*, which is a link to the ACK driver *ack*(U).

Options

–M*n* set maximum identifier length to *n*.

–n do not generate EM register messages.

–NL generate the EM *fil* and *lin* instructions that enable an interpreter to keep track of the current location in the source code.

–w suppress warning messages.

–A enable extra array bound checks, for machines that do not implement the EM ones.

–C do not map upper-case to lower-case.

–u

–U allow underscores in identifiers. It is never allowed to start an identifier with an underscore.

–a do not generate code for assertions.

–c allow C-like strings. This option is mainly intended for usage with C-functions.

–d allow the type ‘‘long.’’

–i*n* set the size of integer sets to *n*.

–s allow only standard Pascal. This disables the **–c** and **–d** options. Furthermore, assertions are not recognized at all (instead of just being skipped).

Diagnostics

All warning and error messages are written on standard error output. Descriptions of run-time errors are read from */profile/module/ack/etc/pc_rt_errors*.

Files

/profile/module/ack/lib/front/em_pc – the Pascal front end.

See Also

libpc(L), *Pascal Reference Manual* in the Programming Guide.

Name

pdump – dump process descriptor (checkpoint) and stack traces

Synopsis

```
pdump core-file
```

Description

Pdump expects a process descriptor (also known as checkpoint) in the *core-file*, and prints it out in symbolic form on *stderr*. Because parts of a process descriptor are architecture-dependent (especially the thread descriptors), there is a variant for each supported architecture, plus an architecture-independent variant which prints certain parts of thread descriptors as uninterpreted hex bytes.

If *core-file* is ‘‘-’’ then *pdump* reads from *stdin*. This feature is only intended for use by programs.

Because Amoeba executable files begin with a process descriptor, they can be dumped as well; no stack trace is printed in this case.

The program deduces which architecture the process descriptor is for based on information in the process descriptor and attempts to analyze the thread descriptors and any stack traces accordingly.

Examples

```
pdump /bin/sh/pd.i80386
```

might print:

```
Process descriptor from architecture: i80386

  offset     addr+     len=     end    flags (5 segments)
000000e0 20000000+00019195=20019195 00000110 fixed read-only text
00019275 2001a000+000013b8=2001b3b8 00000320 fixed read/write data
         2001c000+0000d7f0=200297f0 00000321 heap read/write data
         80000000+00000000=80000000 80000322 stack read/write data
0001a62d 00000000+00004a7b=00004a7b 60000000 symbol table
1 threads:
THREAD 0:       state   00000000
        pc      20000000
        sp              0
```

Assuming a checkpoint of an Intel 386 process exists on the file */dev/dead/16*, this command might print its contents:

```
pdump /dev/dead/16
```

with possible output:

```
Process descriptor from architecture: i80386

  offset     addr+     len=     end    flags (7 segments)
0000017c 20000000+0000d000=2000d000 00000110 fixed read-only text
0000d17c 2000d000+00001000=2000e000 00000320 fixed read/write data
0000e17c 2000e000+0000b000=20019000 00000321 heap read/write data
0001917c 7fffc000+00004000=80000000 00000322 stack read/write data
0001d17c 20021000+00008000=20029000 00000321 heap read/write data
0002517c 20031000+0000e000=2003f000 00000321 heap read/write data
0003317c 20047000+00004000=2004b000 00000321 heap read/write data
1 threads:
THREAD 0:       state   00000005        RUN EXC
        signal       -10        Bad system call or parameters

cs:eip                2b:20007126
ss:esp                23:7ffffc74
eflag                246
ebp                    0
ds/es/fs/gs           23       23       23       23
eax/ebx/ecx/edx ffffffff ffffffff ffffffff 7ffffc9c
edi/esi         ffffffff ffffffff
cr3              1ff8000
faultaddr       ffffffff
code             1ff8000

7ffffcb4: 20007126()
7ffffcd0: 20001ad1()
7ffffce0: 20001dd3()
7ffffcfc: 20000cd2()
7ffffcfc: called from 20000024
```

If there is more than one thread then a register dump and stack trace is provided for each thread.

The last section of information at the end of each thread represents the stack trace. The number before the colon (:) is the stack pointer and the number to the right of the colon is the program counter followed by the arguments with which the function was called in parentheses. Converting the program counters to symbolic names can be done using *nm*(U).

Note that the format of the stack trace information varies per architecture.

File Formats

The input must begin with a process descriptor in network byte order; whatever comes after this is ignored.

See Also

ax(U), nm(U), stun(U).

Name

ppload – show processor load and up-time statistics

Synopsis

```
ppload [-t timeout] [-v] [directory]
```

Description

This program shows some statistics of the (pool) processors. For each processor, it prints its name, the average number of runnable threads during the last few seconds, the potentially available CPU power when the processor is idle (in MIPS), the free memory (in Mbytes), and the time the processor is up.

The optional directory argument specifies an alternate directory where to look for processor names. The default is specified in *ampolicy.h* as `HOST_DIR` (typically */profile/hosts*).

Options

–t timeout — Specifies the timeout (in milliseconds) used when polling a processor. The default is 2000.

–v — This adds a second line for each processor, listing the kernel version string returned by the kernel running on the processor.

Examples

```
ppload
```

might print:

```
zoo00    load 0.00, 36.4 Mips, 27.1 Mb free, up 2 hrs 46 min
zoo01    load 0.20, 36.4 Mips, 26.5 Mb free, up 2 hrs 46 min
```

An example giving verbose output:

```
ppload -v
```

which might print:

```
zoo00    load 0.00, 36.4 Mips, 27.1 Mb free, up 2 hrs 47 min
         Amoeba 5.3 #22 <sun> by versto@splash.cs.vu.nl (Wed Mar  1 13:24:32 MET 1995)
         in directory /amconf/conf0/amoeba/sparc.sun/kernel/sun4m/pool
zoo01    load 0.00, 36.4 Mips, 26.5 Mb free, up 2 hrs 47 min
         Amoeba 5.3 #22 <sun> by versto@splash.cs.vu.nl (Wed Mar  1 13:24:32 MET 1995)
         in directory /amconf/conf0/amoeba/sparc.sun/kernel/sun4m/pool
```

See Also

aps(U), kstat(A).

Name

pr – print a file

Synopsis

```
pr [options] [-columns] [+page] [file] ...
```

Description

Pr formats one or more files for printing. The result is written to *stdout*. If no files are specified, *stdin* is printed. Options are provided for setting the width and height of the page, the number of columns to use (default 1), and the page to start with, among others.

Options

–*columns*
: Format the output in *columns* columns.

+*page*
: Start at page number *page*.

–h *string*
: Use *string* as the page header. Note that if it contains spaces or other characters special to the shell then they should be quoted.

–l*n* Sets page length to *n* lines.

–n Number the output lines.

–t Do not print page header or trailer.

–w*n* Sets line length to *n* characters.

Examples

```
pr -w72 -l60 file
```

Use 72 characters per line and 60 lines per page.

```
pr -3 file
```

List file three columns to a page.

```
pr +4 file
```

Start printing with page 4.

Name

prep – prepare a text file for statistical analysis

Synopsis

```
prep [file]
```

Description

Prep strips off most of the *troff/nroff* commands from a text file and then outputs all the words, one word per line, in the order they occur in the file. This file can then be sorted and compared to a dictionary (as a spelling checker), or used for statistical analysis.

If the *file* argument is not present it reads from *stdin*.

Example

```
prep infile > outfile
```

Name

printenv – print out the current string environment

Synopsis

```
printenv
```

Description

Printenv can be used to print out the string environment of a program, mostly the shell.

Example

```
printenv
```

This might print

```
TCP_SERVER=/profile/cap/tcpipsvr/vmesc1
SPMASK=0xff:0x2:0x4
DISPLAY=klepper:0
ARCHITECTURE=mc68000
HOME=/home
PATH=:/home/bin:/bin:/profile/util:/profile/module/ack/bin
USER=versto
TERM=xterm
SHELL=/bin/sh
_WORK=/home
```

Name

put – put a capability-set into a new directory entry specified by name

Synopsis

```
put [-f] pathname
```

Description

Put reads a capability-set from standard input and uses *sp_append(L)* to enter it into the directory server under the specified *pathname*. Normally, the path name must not already exist in the directory server, but the use of **–f** causes any existing capability-set entered under that name to be replaced using *sp_replace(L)*. The object(s) referred to by any existing capability-set are not destroyed when it is replaced.

If the input does not look like a capability-set or the path name cannot be created in the directory server, a suitable error message will be printed. No check for the validity of the capabilities in the set is performed.

Example

```
get /home/oldname | put /home/newname
```

will install */home/newname* in the directory server as a synonym for */home/oldname*.

See Also

chm(U), del(U), dir(U), get(U), getdelput(U), mkd(U).

Name

pwd – print working directory

Synopsis

```
pwd
```

Description

Pwd prints the full path name of the current working directory.

Name

ref – display a C function header

Synopsis

```
ref function_name
```

Description

Ref is a program which looks up the function header of a particular function in any of a series of reference files. These reference files are produced by the *ctags*(U) program.

Ref is used by *elvis'* shift-K command.

Files

The list of files checked includes

/usr/src/lib/refs
/usr/local/lib/refs

See Also

ctags(U).

Name

refont – changes the notation used for fonts

Synopsis

```
refont [flags] files ...
```

Description

Refont reads a text file which contains font selection codes embedded within it, and it writes the same text with a different notation for fonts.

For example, the original *elvis* documentation used Epson-compatible escape sequences to select different fonts. The command

```
refont -b intro.doc > intro.b
```

could be used to make a file that uses overtyping to implement boldface or underlined text.

Options

–b Emit text which uses the ‘‘backspace’’ notation for fonts. Each underlined character will be preceded by an underscore character and a backspace character. Bold characters are set twice, with a backspace in between. The *more*(U) utility understands this notation.

–c Emit text which uses the ‘‘carriage-return’’ notation for fonts. An entire line of text is written, followed by a carriage return instead of a newline. Then a space is sent for each normal character, an underscore is sent for each underlined or italic character, and each boldface character is sent a second time. Many mainframe line printers accept this notation.

–d Emit text which uses *nroff*-style ‘‘dot’’ commands for fonts. This does not work very well.

–e Emit text using Epson-compatible escape sequences for fonts. This is useful as a ‘‘least common denominator’’ for font notations, because this is the only supported notation to use control-character sequences and also distinguish between italics and underlining.

–f Emit text which uses *nroff*’s *\fX* notation for fonts. Underlined text is denoted by *\fU*, boldface by *\fB*, italics by *\fI*, and normal text by *\fR*. This is somewhat useful in conjunction with *elvis’* ‘‘charattr’’ option.

–x Emit text which has had all font information stripped out.

–I When reading text, **–I** tells *refont* to accept any of the above notations for fonts. Without **–I** it will ignore the ‘‘dot’’ command and *\fX* notations; they will be treated as normal text. In other words, without **–I** the only things that could be recognized as font changes are control-character sequences.

–F This causes *refont* to insert form feed characters between input files.

Warnings

Support for the *nroff*-style ‘‘dot’’ commands is not very good.

If the **–I** option is mixed with the **–f** or **–d** options, the resulting output may not be translatable back into its original format. This is because the original text may have included, for example, strings that looked like *\fX* strings which were not meant to cause font changes.

With **–b** or **–c**, both underlining and italics are implemented by overtyping the underscore character with a text character. Since they are represented the same way, the distinction between underlining and italics is lost.

Name

reserve – make host reservation

Synopsis

```
reserve [-# nhosts] [-a arch] [-b command] [-c res-cap]
        [-p pool] [-s start-time] [-t duration]
```

Description

The command *reserve* is the user interface of the host reservation system. The reservation system implements processor access restrictions and it provides support for batch jobs. By means of a host reservation, a user can get *exclusive* access to a specified number of hosts for a given period of time. This is useful for conducting controlled experiments, e.g., measuring the timing of a parallel Orca program.

A host reserveration specifies the number and type of hosts, a duration and an optional start time. It may also include an optional command to be executed once the resources have been acquired. This command will be added to the batch queue. If no batch command is specified, the selected hosts are made available to the user in a private pool, which can be used as needed during the reservation period.

A reservation implies exclusive use of certain resources. During certain hours there may be permanent reservations for the general use of machines. That is, no individual can obtain exclusive access to resources. For example, on Monday to Friday, from 08:00-20:00, it will typically not be possible to reserve the entire system. Such non-exclusive reservations are predefined in a file which is under the control of the system manager.

Whenever a reservation has been made, a capability for the reservation will be returned. The capability can be used to get status information about the reservation or to cancel the reservation when it is no longer needed.

Reservervations are executed as follows. First the reservation system checks that all the requested hosts are still alive. If some hosts are down, it will try to substitute these with other hosts that are still available. If this is not possible, and the reservation was a batch command, the reservation is cancelled because its working is likely to depend on all the requested hosts being available. Otherwise, the hosts selected are reserved by letting them switch to a new capability for the duration of the reservation (which effectively makes them unavailable to other users). The modified host capabilities are stored in a new pool directory, which is created in the user's */profile/reserve* directory. Any processes that were still running on the hosts are killed. If the reservation specified a batch command *cmd*, it is started using a new session server (see *session*(A)):

```
session -a -b /bin/sh -c cmd
```

The batch command is started in the directory where the reservation was made. The path name of the pool directory is passed to the batch command in the environment variable `POOL`.

Options

–# *nhosts*
A total of *nhosts* processors are to be reserved. The default is all the hosts in the directory */profile/reserve/pool*. This directory will typically have a subdirectory of hosts per host architecture.

–a *arch*
Only processors of type *arch* are to be reserved. The default is the whole processor pool specified. More than one **–a** option may be specified.

–b *command*
This specifies that the command *command* be entered into the batch queue. If the **–s** option was specified, the job is run at the time specified. Otherwise it is run at the first opportunity when all the required resources can be obtained. The command will be executed in the same execution environment (working directory, string and capability environment) as the *reserve* command itself.

–c *reservation-cap*
This option cancels the reservation specified by *reservation-cap*. The cancellation of a reservation can occur any time up until the end of the reserved period. If the reservation period has already begun, this command immediately releases all acquired resources.

–p *pooldir*
The processors for the reservation are to be taken from *pooldir*. This directory may contain one or more subdirectories, one for each architecture. The default processor pool is */profile/reserve/pool*.

–s *start-time*
This option specifies an absolute start time for the reservation. The syntax of the time specification is ‘‘`[[mm]dd]hhmm`’’ (optional month and day, required hour and minutes). If this option is not present, the reservation server will schedule the reservation to start as soon as possible. The latter will depend on the allowed reservation periods, and possibly existing reservations. Information about all current reservations can be obtained by means of the command

```
std_status /profile/cap/reserver/default
```

–t *duration*
This option specifies the length of the reservation in minutes. The default duration is 15 minutes. When a batch job terminates, all the resources acquired for the reservation will be released, regardless of whether or not the maximum reservation time has expired. If a reservation was made without specifying a batch job then the resources will be for the exclusive use of the reserver for the entire time-period, regardless of the number of jobs executed. At the end of this period all acquired resources will be released.

Diagnostics

Reserve prints status information when a new reservation is made. This status information includes the start time and period of the new reservation and the names of the hosts selected. If a reservation cannot be made, *reserve* will print an error message describing the problem.

The reservation service sends a mail message to the user upon completion of a reservation.

This message contains diagnostic information like start time, end time and (for batch jobs) exit status. It is also used to report any problems that may have occurred during the host reservation procedure or the construction of the temporary pool directory.

Files and Directories

/profile/reserve/pool
: The default reservation pool directory.

/profile/reserve/res.`id`
: The capability of reservation with number *id*. This capability can be used to get status information, or to cancel the reservation.

/profile/reserve/pool.`id`
: The private pool corresponding to reservation with number *id*. It will be installed for the user during the reservation period. It will be removed by the reservation server afterwards.

/profile/cap/reserver/default
: The capability of the default reservation service.

Environment Variables

`POOL`
: The pathname of the reserved pool passed to batch jobs.

Examples

The command

```
reserve -# 4 -p /profile/gaxpool -a sparc -t 30
```

asks for a new reservation of thirty minutes for four sparc hosts obtained from */profile/gaxpool*. If this can be done succesfully, status information of the following format will be returned:

```
versto: 1: Mon Jul 11 20:00:01 1994, 30 min:  zoo0[0-3]
```

To start a single Orca batch job on four sparc hosts from the default pool */profile/reserve/pool*, the following command can be used:

```
reserve -# 4 -a sparc -b 'gax -1 orcaprg 4 >gax.out 2>&1'
```

which might return the following:

```
versto: 2: Mon Jul 11 20:00:01 1994, 15 min:  zoo0[4-7]
```

Since no time is specified, the batch job is supposed to be ready within the default time limit of 15 minutes. Note that the batch command has to be quoted. Also note that the output is redirected to a file, because otherwise it would be written to the terminal device from which the reservation was made, which may no longer be available the moment the reservation is executed.

To cancel reservation number 2, the following command can be used:

```
reserve -c /profile/reserve/res.2
```

Status about all pending reservations can be retrieved using `std_status`:

```
std_status /profile/cap/reserver/default
```

This might produce the following output:

```
Reservations allowed 7 days in advance, except on:
        Mon 8:00 - 20:00
        Tue 8:00 - 20:00
        Wed 8:00 - 20:00
        Thu 8:00 - 20:00
        Fri 8:00 - 20:00

Hosts that can be reserved:
        zoo0[0-7] zoo1[0-7] zoo2[0-7] zoo3[0-7] zoo4[0-7] zooseq

Current reservations:
         versto: 1: Mon Jul 11 20:00:01 1994, 30 min:  zoo0[0-3]
         versto: 2: Mon Jul 11 20:00:01 1994, 15 min:  zoo0[4-7]
```

See Also

gax(U), session(U).

Name

rev – reverse the characters on each line of a file

Synopsis

```
rev [file] ...
```

Description

Each file is copied to standard output with all the characters of each line reversed, last one first and first one last.

Example

```
rev file
```

Reverse each line in *file*.

Name

rm – remove a file

Synopsis

```
rm [-fir] name ...
```

Description

Rm removes one or more directory entries. If an object has no write permission, *rm* asks for permission (type ‘‘y’’ or ‘‘n’’) unless **–f** is specified.

Note that the underlying object, such as a Bullet file, is not destroyed. Only the directory entry is removed. The underlying object will be removed by the garbage collection system if there are no further references to the object.

Options

–f Forced remove: no questions asked.

–i Interactive remove: ask before removing each entry.

–r Recursively remove. If the entry refers to a directory, *rm* will recursively descend the directory graph and remove objects. The directory will also be removed once it is empty. Be careful when there are cycles in the directory hierarchy.

Examples

```
rm file
```

Remove file.

```
rm -i *.c
```

Remove all objects ending in *.c*, asking before deleting each.

See Also

del(U).

Name

rmdir – remove a directory

Synopsis

```
rmdir directory ...
```

Description

The specified directories are removed. Ordinary files are not removed.

Examples

```
rmdir /home/foobar
```

Remove directory *foobar*.

```
rmdir /home/f*
```

Remove all directories whose name starts with ‘‘f’’ in the directory */home*.

See Also

del(U).

Name

sed – stream editor

Synopsis

```
sed [-n] [-g] [-e edit_script] [-f script_file] [edit_script]
[file]
```

Description

Sed is a stream editor. It takes an edit script either as an argument or one or more **–e** options (see below) or the *script_file* of the **–f**option and executes it on the *file* specified. If none is specified it reads from *stdin*. Output is produced on *stdout*. The input file is not changed by *sed*. With some minor exceptions, *sed* scripts in Amoeba are like those of UNIX.

Options

–e script
: Execute the specified edit script. More than one **–e** option may be given on the command line. These will be executed on the input in the order that they are given on the command line. It is a good idea to put the *script* between single quotes on the command line to avoid the shell interpreting any character sequences before handing them to *sed*.

–f script_file
: The following argument is a file containing the edit script.

–g Set the global flag on for all **s** commands.

–n Suppress all output not explicitly requested by **p** or **w** commands.

Examples

```
sed -f script file
```

Run the sed script in the file *script* on the contents of *file*. The result is printed on *stdout*. Remember that the input *file* is not modified by *sed*.

```
sed '/pig/s//hog/g' < foo
```

Replace *pig* by *hog* in the file *foo* and print the result on *stdout.*

```
sed -e '/pig/s//hog/g' foo
```

This is functionally identical to the previous example.

Name

sess_setout – change the *stdout* and *stderr* of the current session server

Synopsis

```
sess_setout new-stdout
```

Description

Sess_setout changes the capability for *stdout* and *stderr* to *new-stdout* for the current *session*(U) server. This also changes the capability stored in */dev/console*. The capability *new-stdout* must refer to either a file or a tty device. The name */dev/tty* is recognized as special and changes the *session* server's *stdout* to be the current tty.

This program is useful for debugging the *session* server and for redirecting the *session* server's output to a console window when using X windows.

Diagnostics

`no session svr available`: means that the capability environment variable `_SESSION` was not present.

Environment Variables

`_SESSION` – the session server's capability. It must be present in the capability environment.

Files

/dev/console This is updated by this command. It is used by the *notify*(U) command to send messages to users who are currently logged in.

Example

```
sess_setout /dev/tty
```

causes the *session* server to print further output on the current tty/window.

See Also

notify(U), session(U).

Name

session – session server for POSIX emulation and process management

Synopsis

```
session [-a|-p] [-b] [-c capname] [-d] [-f] [command [arg] ...]
```

Description

The *session* server is needed by most user applications. It is normally started when a user logs in and remains present for the duration of the user's login session. The *session* server provides many of the services needed by the POSIX emulation library (known as *Ajax*). With the help of the *session* server, this library implements pipes, process management, shared file descriptors and */dev/null*. Some POSIX functions do not use the *session* server or use it only when necessary. For example, *read* and *write* only use the *session* server when the file descriptor is shared.

The *session* server provides a *personal* POSIX emulation: each user runs a separate *session* server and users' *session* servers do not communicate. Process IDs are only unique per *session* server. Users cannot kill each others' processes using *kill*(U) since the processes belong to different *session* servers. As a consequence, the *session* server does not need to maintain user IDs or group IDs per process. The access granted to a process depends solely on the capability it presents for an object. In practice this is determined by the directory graph reachable from the user's root directory.

There are several ways to use the *session* server. The normal way is to create a session by specifying the **–a** option and a command to execute. A command consists of all the arguments after the options, if any. It is used as the argv argument to *exec_file*(L). The command (for example, a shell) may spawn sub-processes which also belong to the session. When started this way the *session* server will publish its server capability under */dev/session*, the null device capability under */dev/null* and the process directory */dev/proc* which contains the capabilities for all the processes created under the session. If none of the **–a**, **–p** or **–c** options are specified then the various capabilities are not published.

There are very few uses for the server without the **–a** option. (It is not the default for historical reasons.) It can be used this way only to start a process that must run within its own session. If **–c** is not specified then the capability for the process will be inaccessible to the user and it will not be possible to stun the process, although it can receive signals via the interrupt key from the terminal. It is strongly recommended that the **–a** option always be specified.

One minute after the last process in the session finishes, the *session* server exits and any published capabilities are removed. This typically occurs when a user logs out. During the minute wait, new processes can be started which attach themselves to the *session* server. For example, if the user logs out and then logs in again within one minute, then the previous *session* server will continue to be used.

To add a process to a session, the *session* server capability (taken from */dev/session*) should be passed as `_SESSION` in the capability environment when the process is started. A shell already running under a *session* server does this automatically. *Ax*(U) also does this by default if it can find */dev/session*, so normally it is not necessary to worry about this.

Options

The session server recognizes the following options:

−a Attach mode preserves an existing session if one exists, but will create a new one if no (usable) session exists. If the name under which the server capability is to be published already exists and a server is responding to the capability found under that name, the new *session* server normally refuses to run. If it is started with **−a** then it hands over any command it has to the extant server and exits. This is used to ensure that users, no matter how many times they login, always have a single *session* server.

−b By default, if a command is given as an argument to the *session* server , the name of the command will be prefixed with a '-'. This is useful in case the command is a login shell, which will typically perform extra initialization (e.g., the user's *profile*) as a result of this. Specifying the **−b** option causes the command name to be passed unchanged.

−c capname Specifies the name where the server capability is published. The default is */dev/session*. This option also forces the publication of the *session* server's various capabilities.

−d Debugging. Turns on extensive debugging output (unless the program is compiled with NDEBUG defined). The output is unlikely to be helpful for users.

−f Force publication of the capability even if a server is still responding to the currently published capability. This option is incompatible with the **−a** option.

−p Make the server permanent. Normally the server exits 1 minute after the last process using its services has terminated. This option prevents this.

Environment Variables

`_SESSION` The *session* server's server capability is exported to the capability environment of the *command* under this name.

Files and Directories

/dev/session Default server capability (use **−c** option to override).

/dev/null Null device server.

/dev/proc This directory is implemented inside the *session* server. Process capabilities for all processes known to the *session* server are stored here, with their process ID as object name. Process ID 1 is the *session* server. These capabilities can be presented to *stun*(U) to stun them. When a process terminates, the *session* server removes its capability from */dev/proc*. Note that to kill a *session* server it is better to do

```
std_destroy /dev/session
```

than to use *stun* since the *session* server can then clean up properly before exiting.

/dev/dead If this directory exists, the corpses of processes that died because of an

exception or were killed by SIGQUIT are embalmed and put on display in this directory, with their process ID as filename. Such dumps can be inspected with *pdump*(U).

/dev/console This is the write-only capability for *stdout* and *stderr* for the *session* server where it writes any messages. Typically it is the capability for a tty device although it is permitted to be a file. It can be used for sending messages to system users (see *notify*(U)). Per default the session owner and the members of the groups in which the session owner is registered can write this device. If the latter is not desired then the */dev/console* permissions can be modified with *chm*(U). This can be done in your *.profile*. The */dev/console* capability can be altered for debugging purposes or when logged in to an X windows session. See *sess_setout(U)* for details.

Limits

- The process table has space for 300 processes.
- Process IDs range from 2 to 32767 inclusive (and are reused).
- The pipe server uses two threads per open pipe. This makes pipe creation slow. Further, memory shortage may limit the number of threads that can be created and therefore the number of pipes.

Programming Interface Summary

There are two aspects to the programming interface of the *session* server. One is the commands that it accepts via the RPC mechanism and the other is the set of POSIX functions it supports. The RPC interface to the *session* server is defined in the file *src/h/class/sessvr.cls*. Below is a short list of the POSIX functions for which the *session* server is required. The list is indicative, not complete.

The *session* server is required for the following POSIX process management functions: fork, execve and family, wait, sigaction, kill, getpid, getppid, getpgrp, setpgrp, pipe (see *posix*(L)), by the Standard C function signal (see *stdc*(L)) and by the Amoeba functions newproc and newprocp (see *newproc*(L)).

Administration

Most users will start a *session* server when they login. This is best done from the */Environ* file. When a *session* server dies, the directory */dev/proc* will disappear with it. Any processes that were still registered there but do not need the *session* server for their continued survival may continue to run to completion. These orphans can no longer be killed by the user but only by someone with the */super* capability. A new incarnation of the *session* server will not find and adopt the orphan processes. This presents a small cleanup problem and system administrators should regularly tidy up processes with no valid owner capability.

Examples

To start a permanent *session* server from UNIX, with its output ignored:

```
ax -n /profile/util/session -a -p
```

This command returns as soon as the server process is started.

To start a temporary *session* with a shell running in it:

```
ax /profile/util/session -a /bin/sh
```

This returns when the shell returns.

See Also

ax(U), gax(U), exec_file(L), newproc(L), pdump(U), posix(L), sess_setout(U), stdc(L), stun(U).

Name

sh – shell

Synopsis

```
sh [-acefiknqstx] [file]
```

Description

Sh is a command language interpreter. It is known as *the shell*. It is derived from the MINIX shell and is compatible with the UNIX version 7 Bourne shell. It implements I/O redirection, pipes, magic characters, background processes, shell scripts and most of the other features of the V7 Bourne shell. A few of the more common commands are listed in the examples section, followed by a short explanation. Below is a description of the command syntax and the built-in commands.

Command syntax

The '#' character begins a one-line comment, unless the '#' occurs inside a word. The tokens ';', '|', '&', ';;', '||', '&&', '(', and ')' stand by themselves. A *word* is a sequence of any other non-white space characters, which may also contain quoted strings (quote characters are '´', '"', '`', or a matching '${ }' or '$( )' pair). A *name* is an unquoted word made up of letters, digits, or underscores ('_'). Any number of white space characters (space and tab) may separate words and tokens.

In the following syntax, '{ ... }?' indicates an optional thing, '{ ... }*' indicates zero or more repetitions, '{ ... | ... }' indicates alternatives. Finally, '*char*' is used when character *char* is meant literally. Keywords are written in bold format.

statement:

'(' *list* ')'
'{' *list* ';' '}'
for *name* { **in** { *word* }* }? **do** *list* ';' **done**
{ **while** | **until** } *list* ';' **do** *list* ';' **done**
if *list* ';' **then** *list* ';' { **elif** *list* ';' **then** *list* ';' }* { **else** *list* ';' }?**fi**
case *name* **in** { *word* { '|' *word* }* ')' *list* ';;' }* **esac**
function *name* '{' *list* ';' '}'
name '()' '{' *list* ';' '}'
time *pipe*
Redirection may occur at the beginning or end of a statement.

command:

{ *name*'='*word* }* { *word* }*
This creates a temporary assignment to variable *name*, during the executing the second part. Redirection may occur anywhere in a command.

list:

cond
cond ';' *list*
cond '&' *list*

cond:

pipe
pipe '&&' *cond*
pipe '||' *cond*

pipe:
statement { '|' *statement* }*

Shell Variables

The shell defines several variables and allows others to be defined.

The following standard special variables exist and are automatically set by the shell:

! The process id of the last background process.

The number of arguments available.

$ The process id of the shell.

– The flags given to the shell when it started or by the last set command.

? The value returned by the last foreground command.

In addition any positional arguments are specified by a single digit. Positional arguments are assigned using *set*. Other variables are assigned by

```
name=value
```

The following variables are used by the shell and may be set by the user if they are not read-only.

IFS *Internal field separator*, used during substitution and the *read* command. It is normally set to *space*, *tab* and *newline*.

HOME The default directory for the *cd* command.

PATH The search path for executable commands and **.**'d files.

PS1

PS2 PS1 is the primary prompt for interactive shells. The prompt $PS2 is printed when a statement is not yet complete, after having typed return.

SHELL This is the name of the executable for the shell. This is typically */bin/sh* for this shell. It is used for invoking shell command scripts.

Variable Substitution

A variable is substituted for its value by preceding the variable name with a dollar symbol.

$*variable*

Other constructions may also be used:

${*variable*}
The value of the variable, if any, is substituted. The braces are only required to distinguish the variable name from leading or trailing letters, digits or underscores which would otherwise form part of the name. If *variable* is '*' or '@' then all the positional parameters starting from $1 are substituted, separated by spaces.

Note that in the following *word* is not evaluated unless it is used.

${*variable*−*word*}
If *variable* is set its value is substituted. Otherwise *word* is substituted.

${*variable*=*word*}
If *variable* is not set then *variable* is set to *word* and then the value of *variable* is substituted.

${*variable*?*word*}
If *variable* is set then its value is substituted. Otherwise *word* is printed and the shell exits. If *word* is null an error message will be printed.

${*variable*+*word*}
If *variable* is set then substitute *word* else substitute the null string.

Command Substitution

The shell will substitute commands between two grave accents (`) with the standard output produced by those commands. Trailing newline characters are removed from the standard output. Only interpretation of backslashes (\) is done before interpretation of the command string.

To nest command substitution, the grave accents can be quoted using backslashes.

Expressions

Expression may contain alpha-numeric variable identifiers and integer constants and may be combined with the following operators: == != <= < > >= + - * / % ! ( )

Command execution

After evaluation of keyword assignments and arguments, the type of command is determined. A command may execute a shell built-in or an executable file.

Any keyword assignments are then performed according to the type of command. Assignments in built-in commands marked with a † persist, otherwise they are temporary. Assignments in executable commands are exported to the sub-process executing the command.

There are several built-in commands.

: Only expansion and assignment are performed. This is the default if a command has no arguments.

. *file* Execute the commands in *file* without forking. The file is searched in the directories of $PATH. Passing arguments is not implemented.

break [*levels*]

cd [*path*] Set the working directory to *path*. A null path means the current directory. If *path* is missing, the home directory (`$HOME`) is used. If *path* is **..**, the shell changes directory to the parent directory, as determined from the value of `_WORK`.

continue [*levels*]

echo ... *Echo* prints its arguments followed by a newline. If the **−n** argument is given the newline is not printed.

eval *command ...*

exec *command arg ...*

The executable command is executed without forking. If no arguments are given, any I/O redirection is permanent.

exit [*status*]

read [**−r**] [**−u***n*] *name ...*

Read the value of variables specified from standard input. The **−u** option can be used to specify a different file descriptor. The first variable name may be of the form *name***?***prompt*.

readonly [*name ...*]

Make the variables specified read-only, i.e., future assignments to them will be refused.

set [±[*a-z*]]

Set (**−**) or clear (**+**) a shell option:

−a allexport	all new variable are created with export attribute
−e errexit	exit on non-zero status
−k keyword	variable assignments are accepted anywhere in command
−n noexec	compile input but do not execute (ignored if interactive)
−f noglob	do not expand filenames
−u nounset	dollar expansion of unset variables is an error
−v verbose	echo shell commands on stdout when compiling
−h trackall	add command path names to hash table
−x xtrace	echo simple commands while executing

set [±**o** *keyword*] ...

Set (**−**) or clear (**+**) shell option *keyword*:

set [**−−**] *arg ...*

Set shell arguments.

shift [*number*]

test *expression*

Test evaluates the *expression* and returns zero status if true, and non-zero status otherwise. It is normally used as the controlling command of the *if* and *while* statements. See test(U) for a complete description.

trap [*handler*] [*signal ...*]

umask [*value*]

wait [*process-id*]

If *process-id* is given, wait for the specified process to finish. Otherwise wait for any process to finish.

Examples

```
date                            # Regular command
sort <file                      # Redirect input
sort <file1 >file2              # Redirect input and output
cc file.c 2>error               # Redirect standard error
a.out >f 2>&1                   # Combine standard output and standard error
sort <file1 >>file2             # Append output to file2
sort <file1 >file2 &            # Background job
(ls -l; a.out) &                # Run two background commands sequentially
sort <file | wc                 # Two-process pipeline
sort <f | uniq | wc             # Three-process pipeline
ls -l *.c                       # List all files ending in .c
ls -l [a-c]*                    # List all files beginning with a, b, or c
ls -l ?                         # List all one-character file names
ls \?                           # List the file whose name is question mark
ls '???'                        # List the file whose name is three question marks
v=/usr/ast                      # Set shell variable v
ls -l $v                        # Use shell variable v
PS1='Hi! '                      # Change the primary prompt to Hi!
PS2='More: '                    # Change the secondary prompt to More:
ls -l $HOME                     # List the home directory
echo $PATH                      # Echo the search path
if ... then ... else ... fi     # If statement
for ... do ... done             # Iterate over argument list
while ... do ... done           # Repeat while condition holds
case ... esac                   # Select clause based on condition
echo $?                         # Echo exit status of previous command
echo $$                         # Echo shell's pid
echo #                          # Echo number of parameters (shell script)
echo $2                         # Echo second parameter (shell script)
echo $*                         # Echo all parameters (shell script)
```

Name

shar – shell archiver

Synopsis

```
shar file ...
```

Description

The named files are collected together into a shell archive and written onto standard output. The individual files can be extracted by redirecting the shell archive into the shell. The advantage of *shar* over other archiving tools is that *shar* archives can be read on almost any UNIX system, whereas numerous, incompatible versions of other archiving tools are in widespread use. Extracting the files from a shell archive requires that *sed*(U) be available.

Examples

```
shar *.c > s
```

Collect C programs in shell archive *s*.

```
sh < s
```

Extract files from shell archive *s*.

See Also

sed(U).

Name

size – print the size of the text, data, and bss segments of a program

Synopsis

```
size [file] ...
```

Description

Size is used to obtain information about the size of executable binary files. The text, data, bss, and total sizes for each *file* argument are printed in hexadecimal. If no arguments are present, the file name *a.out* is assumed. The amount of memory available for combined stack and data segment growth is printed in the column ‘‘stack.’’ The total amount of memory allocated to the program when it is loaded is printed in decimal under ‘‘dec.’’ This value is just the sum of the other four columns. The hexadecimal equivalent of ‘‘dec’’ is listed under ‘‘hex.’’

Example:

```
size /bin/ls/pd.sparc
```

might print

```
text    data    bss     stack   dec     hex
e000    2000    28148   4000    213320  34148
```

and

```
size /bin/ls/pd.sparc /bin/date/pd.i80386
```

might print

```
text    data    bss     stack   dec     hex
e000    2000    28148   4000    213320  34148   /bin/ls/pd.sparc
c000    2000    11ad8   4000    113368  1bad8   /bin/date/pd.i80386
```

Name

sleep – suspend execution for a given number of seconds

Synopsis

```
sleep seconds
```

Description

The caller is suspended for the indicated number of seconds. This command is typically used in shell scripts.

Example

```
sleep 10
```

Suspend execution for 10 seconds.

Name

sort – sort a file of ASCII lines

Synopsis

```
sort [-bcdfimnru] [-tx] [-o name] [+pos1] [-pos2] [file ...]
```

Description

Sort sorts the contents of one or more files on a per line basis. If no files are specified, *stdin* is sorted. Output is written on standard output, unless **–o** is specified. The default sort key is an entire line. The options **+pos1 –pos2** use only fields *pos1* up to, but not including *pos2* as the sort key, where a field is a string of characters delimited by spaces and tabs, unless a different field delimiter is specified with **–t**. Both *pos1* and *pos2* have the form *m.n* where *m* tells the number of fields and *n* tells the number of characters. Either *m* or *n* may be omitted.

Options

–b Skip leading blanks when making comparisons.

–c Check to see if a file is sorted.

–d Dictionary order: ignore punctuation.

–f Fold upper case onto lower case.

–i Ignore non-ASCII characters.

–m Merge presorted files.

–n Numeric sort order.

–o *name*
Use *name* as the output file. Default is *stdout*.

–r Reverse the sort order.

–t Following character is field separator.

–u Unique mode: delete duplicate lines after sorting.

Examples

```
sort -nr file
```

Sort keys numerically, reversed.

```
sort +2 -4 file
```

Sort using fields 2 and 3 as key.

```
sort +2 -t: -o out
```

Field separator is ‘‘:’’.

```
sort +.3 -.6
```

Characters 3 through 5 form the key.

See Also

uniq(U).

Name

spell – check the spelling in a document

Synopsis

```
spell [file1 ...]
```

Description

Spell reads the named files (or *stdin* if no arguments are given), deletes *troff/nroff* directives and punctuation, collects all the words and checks them against a dictionary of known words. Any words not found in the dictionary are reported on *stdout*. The spelling used in the dictionary is American English.

Note that words must appear in the dictionary as they appear in the text. Variations due to tense, etc. are not deduced.

Files

The list of known words is in the file */usr/lib/dictionary*.

File Formats

The dictionary consists of words, one per line in alphabetical order.

Name

split – split a large file into several smaller files

Synopsis

```
split [-n] [file [prefix]]
```

Description

Split reads *file* and writes it out in *n*-line pieces. By default, the pieces are called *xaa*, *xab*, etc. The optional *prefix* argument can be used to provide an alternative prefix for the output file names.

Options

–n *n* specifies the number of lines per piece (default: 1000).

Examples

```
split -200 file
```

Split file into pieces of 200 lines each.

```
split file z
```

Split file into *zaa*, *zab*, etc.

Name

starch – backup and restore tool

Synopsis

```
starch -o [options] path ...
starch -o [options] -A path [options]
starch -i [options] [path ...]
starch -t [options] [path ...]
starch -g [options] path ...
```

Description

Starch can save an entire Soap directory graph to an archive file from which it can later restore that same graph. The archive can be written to, and read from, a bullet file, standard input, standard output or a tape. It also performs related functions such as listing the objects in a directory graph and listing the contents of an archive. There are options to skip certain directories or file names matching a given pattern.

The Amoeba directory graph contains cycles. Therefore a simple tree traversal algorithm as is typically used by utilities such as *tar*(U) and *find*(U) can get into an infinite loop. To avoid following cycles, it is necessary to remember at least the capabilities of all the directories encountered on the path to a directory. In fact, *starch* remembers the capabilities of all objects it has encountered, so it will not only avoid cycles but in general archive each object at most once.

Options

Exactly one mode option must be specified, chosen from the list below. The interpretation of path name arguments depends on the mode. The archives are read from *stdin* and written to *stdout*, unless a **–f**, **–l**, **–m** or **–M** option is present. The mode options are:

–i Input mode. *Path* names specify starting points of subsets of objects to be restored; if no path names are specified, the entire archive is restored.

–g Name generation mode. This does not manipulate archives. It traverses one or more directory graphs whose starting point(s) are given as arguments and prints the names of the objects found. These are exactly the objects that would be written to the archive in output mode. The same object may be listed under multiple names.

–o Output mode. One or more path names indicating the starting point(s) of the directory graph(s) to be written to the archive file must be specified. Starting point arguments are not allowed when the **–A** option is used to create an incremental dump.

–t Tabulation mode. This lists the contents of the archive rather than restoring from it; it is otherwise similar to input mode.

Archive selection and handling options which are valid in combination with **–i**, **–o** and **–t** are:

–b *N*

The number *N* indicates the size in bytes of the read or write operations used to access the archive. This option can be useful while writing tapes. The number *N* can be

followed by `b`, `k`, `m` or `g` to indicate respectively 512-byte blocks, kilobytes, megabytes and gigabytes. The upper case versions of these suffices are also recognized and have the same meaning as the lower case versions.
The number *N* must be a multiple of 1024, the internal block size of *starch*.

–f *path*
Use the indicated *path* to read/write the archive. Multiple **–f** options can be used to read multiple archives with the **–i** and **–t** option. This is used when reading base archives in combination with incremental archives (see **–a** and **–A** below). The restored objects will have the creation time of the archive indicated by the first **–f** option.
One of the options **–l**, **–m** or **–M** passed as argument before the **–f** can be used to change the device type of the archive.

–l Use the *vdisk*(L) stub interface to read/write the archive mentioned with the next **–f**, **–a** or **–A** option. If **–l** is not followed by a **–f**, **–a** or **–A** and the `FLOPPY` environment variable is defined, then *starch* acts as if an **–f** `$FLOPPY` option was added to the end of the option list.
When the archive does not fit on a single floppy it is advisable to use the **–s** option to indicate the size of the floppy disk.

–m Use the *tape*(L) stub interface to read/write the archive mentioned with the next following **–f**, **–a** or **–A** option. If **–m** is not followed by a **–f**, **–a** or **–A** and the `TAPE` environment variable is defined, then *starch* acts as if an **–f** `$TAPE` option was added to the end of the option list.
With tape drives that allow writing after the preliminary end-of-tape message, *starch* does not need to know the size of the tape in advance. If error messages appear at the end of the tape, *starch* still will produce readable archives, but in such cases it is preferable to use the **–s** option to indicate the size of the tape.
The tape will be rewound to the load point when the archive has been read or written.

–M As under **–m** but without rewinding the tape.

–n *N*
Indicates the number of buffers used between the main and read or write threads of *starch*. If the number of buffers to use is 1, no buffering will take place and only a single thread will be used. Only the Amoeba version of *starch* can use multiple threads. The default numbers of buffers under Amoeba is 80.

–s *N*
The number *N* indicates the size, in bytes, of the medium on which the archive is to be written. This option is most useful while writing (floppy) disks or tapes. The same single letter suffices can be used as with the **–b** option.

–S Print the total number of bytes written to, and read from, archives.

Options that can only be used while creating archives with **–o** are:

–a *path*
In addition to the archive which contains the contents of the files and directories (known as the *base* archive), a *description archive* is created and stored under the name *path*. A description archive is a *starch* archive containing the capability and version number for each directory (see *soap*(A)) and one capability for each archived file. The description archive can be used with later invocations of *starch* to make incremental archives (see

the **–A** option). The description archive can also be used to retrieve lost capabilities. *Starch* uses the 'illegal' version number zero for all directories for which it cannot get the sequence number.

One of the options **–l**, **–m** or **–M** given as argument before the **–a** can be used to change the device type of the description archive.

–A *path*

This option is used to make incremental archives. The *path* argument must refer to a description archive created with the **–a** option.

For each directory found in the description archive, *starch* only needs to consider directories with sequence numbers that differ from those in the description archive. These directories are scanned for new capabilities. New directory capabilities are followed. The archive it creates contains the whole new directory structure and any new capabilities and bullet files. This type of archive is called *incremental*. *Starch* will only work properly if the same **–U**, **–Y**, **–Z** **–u**, **–y** and **–z** options which were used while making the base archive are used to make the incremental archive. If any of these options have changed since the last base achive was made then make a new base archive.

Individual files can be restored from incremental archives. To restore complete archives, use multiple **–f** options with the **–i** option. The directory structure restored is that of the archive mentioned in the first **–f** option.

For obscure reasons having to do with capability-sets, *starch* will sometimes write bullet files on an incremental archive that are already present in the base archive.

–A and **–a** can be combined to create levels of incremental archives.

One of the options **–l**, **–m** or **–M** given as argument before the **–A** can be used to change the device type of the archive.

Options that can be used while restoring archives with **–i** are:

–B *path*

All bullet files to be created by *starch* will be created on the bullet server specified by *path*.

–D *path*

All directories to be created by *starch* will be created on the directory server specified by *path*.

–k *path*

This option is useful when the archive is somehow damaged. All bullet files, capabilities and directories that are not accessible through the entry points due to missing directory information are stored in the directory specified by *path* under their *id* number. More information on the *id* number is given in the section ''Informative Output Format'', below. This option is not valid in combination with partial restores.

–r Retain. Normally *starch* stops when an entry it tries to create already exists. With this option *starch* will leave all existing entries intact and only create new entries.

–w Overwrite. Normally *starch* stops when an entry it tries to create already exists. With this option *starch* will replace existing entries with the entry stored in the archive.

The remaining options may be combined with any of the mode options:

–v Verbose. This may be used to increase the amount of output generated. There are four

levels of verbosity: silent, names only, names and status information, and extended status information. Tabulation and name generation mode (**–t** and **–g**) start at the second level (names only); the other modes are silent by default (except for error messages).

–x *path*
Specifies a path name which should not be archived/restored/listed/etc. (as appropriate for each mode). This may be repeated to specify multiple exceptions. If a directory is specified in this way, the subgraph starting at *path* will not be traversed. Other names for the same object will be treated as links (thus, the object itself will not be copied to the archive at all).

–X *pattern*
Objects whose last path name component match *pattern* are skipped. The pattern matching syntax is (hopefully) identical to that of the shell (so it will need to be quoted to stop the shell trying to interpret it). If a directory is skipped, the subgraph starting at that point will not be traversed. Unlike the **–x** option, objects that are skipped through this option may still be reached by other names. This option may be repeated to specify multiple exceptions.

Selecting Files on Input

Starch has a rather complex method for selecting files to be listed or archived to tape. There are three ways to handle an entry: ignore it, archive/restore the capability of the object, archive/restore the contents of the object. The only type of objects *starch* can archive and restore the contents of, are bullet files and directories. Apart from the **–x** and **–X** options mentioned above, there are two mechanisms for specifying how to handle an entry. One is based on the port number in capabilities and the other is based on the first character returned by *std_info*(U). The first selection method used is the one based on capabilities. If the port of a capability is identical to one of the ports of the capabilities stored in any of the paths mentioned by the **–U**, **–Y** or **–Z** options, the entry is handled as specified by the option. If no port match is found, the second method, based on *std_info*, is used. The default method is to select every entry that is a bullet file (*std_info* string begins with ‘‘–’’) or a directory (*std_info* string begins with ‘‘/’’) and to archive the capability for all other types.
The options:

–u *object_types*
Specifies the *std_info* types of the objects for which the contents are to be archived. Only directory and bullet types may be specified with this option. The default is ‘‘/–’’. To disable matching with *std_info* strings use **–u** "".

–y *object_types*
Specifies the *std_info* types of the objects for which the capabilities are to be archived. If this option is present, the default method for handling objects of unknown type changes from storing the capability to ignoring the object. The **–z** and **–Z** options can not be combined with this option.

–z *object_types*
Specifies the *std_info* types of the objects to be ignored. The **–y** and **–Y** options can not be combined with this option.

–U *path*

Specifies the ports of the objects for which the contents are to be archived. This option may be repeated. Only directory and bullet ports may be specified with this option.

–Y *path*

Specifies the ports of the objects for which the capabilities are to be archived. This option may be repeated. If this option is present the default method of handling objects of unknown type changes from storing the capability to ignoring the object. The **–z** and **–Z** options can not be combined with this option.

–Z *path*

All entries with their port identical to the port of the capability stored under *path* will be ignored. This option may be repeated. The **–y** and **–Y** options can not be combined with this option.

Informative Output Format

Output in tabulation and name generation modes is presented on *stdout*. Output in all other modes is presented on *stderr*. The output of *starch* in tabulation mode sometimes uses *id* numbers to identify objects. These numbers are used for the internal administration of *starch* and have no relation to the object numbers in capabilities. The output in tabulation mode is intended to be self explanatory.

Path name arguments

Starch is not good at handling ‘‘.’’ and ‘‘..’’ in path name arguments. Specifying a single ‘‘.’’ in output mode to indicate that the current directory should be archived will work. Specifying ‘‘..’’ in argument path names will lead to confusion. In output and list mode the argument path names are normalized (see *path*(L)). Normalization is not done for input and tabulation mode because the source archive might contain ‘‘.’’ or ‘‘..’’ entries.

Diagnostics

The program prints an error when a header cannot be parsed or an I/O error occurs on the archive file. If this happens during tape writes, the user is offered a chance to insert a new tape, retry the write or abort. If an I/O error on the archive happens during a restore, the user is offered a chance to insert a new tape, retry the read, abort or assume that the end of the archive has been reached. In the last case *starch* will attempt to restore as much as it can. The **–k** option can be useful under such circumstances.

Environment Variables

TAPE – specifies the tape server when the **–m** or **–M** option is not followed by a **–f**.

FLOPPY – specifies the floppy drive to use when the **–l** option is not followed by a **–f**.

Warnings

Due to a missing interface in the Soap Server, it is not possible to restore objects’ modification times from archives (the time is put on the archive however, so if that interface is ever added...).

The **–v** has no effect in combination with **–g**.

Inaccessible directories are ignored. Often no warning is issued.

Inaccessible bullet files are ignored. A warning is issued.

File Format

The archive file format is unique to *starch*; standard archive formats such as *tar* (which are also available in Amoeba) cannot adequately handle directed graphs and do not provide space for the information about files and directories that must be saved in order to be able to restore them to their original state (such as directory column names and column masks). The exact format is quite complex and can be gleaned from the source code; it should be sufficient to state that all information is split in blocks of 1024 bytes, possibly padded with zero bytes, and headers are byte-order independent.

Examples

To list the file names of everything in directory *src* and below:

```
starch -g src
```

This might print:

```
src
src/main.c
src/test
src/a.out
src/test/testinput
```

To archive the contents of directory *src* to file *src.starch:*

```
starch -o -f src.starch src
```

To get a verbose listing of the contents of file src.starch:

```
starch -tv -f src.starch
```

To restore everything from *src.starch* except the directory *src/test*, and get a listing of the files restored:

```
starch -i -v -x src/test -f src.starch
```

This might print:

```
src
src/main.c
src/a.out
```

To archive to tape everything accessible through */super* on a soap and two bullet servers. This method of specifying the directory and file servers to be archived makes *starch* much faster.

```
starch -mof /super/hosts/machine/tape:00 -u "" -U / \
         -U /super/cap/bulletsvr/bullet0 \
         -U /super/cap/bulletsvr/bullet1 /super
```

To make a base archive on tape and a description archive on file:

```
starch -mof /super/hosts/machine/tape:00 \
        -a dd \
        -u "" -U / \
        -U /super/cap/bulletsvr/bullet0 \
        -U /super/cap/bulletsvr/bullet1 /super
```

The description archive *dd* will be made in the current directory.

To make an incremental archive:

```
starch -mof /super/hosts/machine/tape:00 \
        -A dd \
        -u "" -U / \
        -U /super/cap/bulletsvr/bullet0 \
        -U /super/cap/bulletsvr/bullet1 /super
```

To restore from the base and incremental tapes in the two previous examples, to a file system with only the directory `/super` present:

```
starch -i -mf /super/hosts/machine/tape:00 \
        -mf /super/hosts/machine/tape:00
```

Note that the first tape that should be inserted in the drive is the incremental archive. Once *starch* is finished with the incremental archive it will prompt for the base archive.

See Also

dgwalk(L), find(U), tape(U), tar(U).

Name

std_copy – replicate an object

Synopsis

```
std_copy servercap original newcopy
```

Description

As part of the automatic replication facilities, servers implementing abstract objects (such as files) support the STD_COPY command which requests them to replicate a particular object. This allows a higher degree of availability for an object by storing copies of it on several different servers. The automatic replication facility of the Soap Server is implemented using the STD_COPY command and is the preferred method of replication. However, in some cases it is convenient to be able to execute the command for a particular object. *Std_copy* is a utility which permits the user to directly execute a STD_COPY command. *Servercap* is the path name for the capability for the server which is to make the copy, *original* is the path name for the capability for the object to be copied and *newcopy* is the path name to store the capability for the new copy of the object.

This differs from the standard copy command in that it works for all abstract objects and not just files, and more importantly, it allows a particular server to be specified on which to make the copy.

Diagnostics

Three types of error are possible:

1. Errors while looking up the capabilities of the server and the original object in the directory server.
2. Errors during the actual copy operation itself (for example, insufficient permission to copy the object).
3. Errors while attempting to store the capability of the newly copied object in the directory server.

The error messages for the three different types are clear and include the explanation of the error status.

Warnings

In most cases there is little sense in replicating on the same server as the original object. To increase the availability of the file it is best to select a different server from the one storing the original.

Example

```
std_copy /profile/cap/bulletsvr/b2 /home/myfile /home/myfile2
```

makes a copy of */home/myfile* on the bullet file server known as *b2* and stores the capability for the copy under the name */home/myfile2*

See Also

cp(U), soap(A), std_destroy(U), std_info(U), std_restrict(U), std_touch(U).

Name

std_destroy – destroy an object

Synopsis

```
std_destroy objectcap ...
```

Description

Std_destroy executes an `STD_DESTROY` command for each object in its argument list. If the destroy operation succeeds it also deletes the directory entry for the object. Only servers for abstract objects accept this command. (Servers for physical objects, such as a disk, would need special assistance to implement it.) Most servers require a special right to be set in the capability to execute this command.

The exit status of the command is the number of errors encountered.

Diagnostics

Three types of error are possible:

1. Errors while looking up the capability of the object in the directory server.
2. Errors during the actual destroy operation itself (such as insufficient permission).
3. Errors while attempting to delete the capability for the object from the directory server.

The error messages for the three different types are clear and include the explanation of the error status.

Example

```
std_destroy /home/myfile /home/bin
```

will destroy the two objects */home/myfile* and */home/bin* and delete their capabilities from the directory server.

See Also

rm(U), del(U), std_copy(U), std_info(U), std_restrict(U), std_touch(U).

Name

std_info – get the standard information about an object

Synopsis

```
std_info object ...
```

Description

Std_info prints the standard information string for the objects specified. This information string is the same as printed by the

```
dir -l
```

command.

Several servers have highly abbreviated information strings. The following lists information strings for well known objects:

```
/       Soap directory
%       Soap kernel directory
-       Bullet file
@       Virtual disk
+       Terminal
```

In addition to the character, further information such as the size of the object or the rights are also printed. For more details about the standard information for a server see the manual page for the server.

Diagnostics

Two types of error are possible:

1. Errors while looking up the capability of the object in the directory server.
2. Errors during the actual standard info operation itself.

The error messages for the three different types are clear and include the explanation of the error status.

Example

```
std_info /home/pclient
```

where */home/pclient* is a file might produce

```
-    41184 bytes
```

See Also

dir(U), std_copy(U), std_destroy(U), std_restrict(U), std_touch(U).

Name

std_restrict – produce a version of a capability with restricted rights

Synopsis

```
std_restrict original mask newcap
```

Description

Std_restrict asks the server responsible for the capability *original* to produce a new version of the capability with the rights bits bitwise-anded with the *mask*. It cannot add rights. It can only remove them. The resultant capability is stored under the name *newcap*. The *mask* is interpreted as a hexadecimal number.

Diagnostics

Three types of error are possible:

1. Errors while looking up the original capability in the directory server.
2. Errors during the actual restrict operation itself.
3. Errors while attempting to store the restricted capability in the directory server.

The error messages for the three different types are clear and include the explanation of the error status.

Example

```
std_restrict /home/mine 1f /home/mine.public
```

will produce a capability for the same object as */home/mine* but with only the rights in the bottom 5 bits switched on that were originally switched on in */home/mine*. The capability will be stored in */home/mine.public*.

See Also

rm(U), dir(U), std_copy(U), std_destroy(U), std_restrict(U), std_touch(U).

Name

std_status – get server status

Synopsis

```
std_status server
```

Description

Std_status prints the status information for the server whose capability is given. This is usually a couple of lines, but can be a couple of pages of output. The layout of the server status is documented with each individual server.

Diagnostics

Error messages are self-explanatory.

Example

```
std_status /profile/pool/mc68000/.run
```

might print

```
HOST     STAT SINCE LREPL LPOLL NPOLL NBEST NCONS   MIPS   NRUN  MBYTE
unlk     DOWN 45:02   54s   44s  7460     5     5
bpl      UP   43:49    0s    0s  3577    76   990  0.900  0.000  2.524
bmi      UP   49:06    3s    3s  7403   114   990  0.900  0.000  2.481
blo      UP   40:25    1s    1s 29110    48   774  0.900  0.000  3.119
```

See Also

bstatus(A), std_info(U), std(L).

Name

std_touch – touch an object so that the server does not garbage collect it

Synopsis

```
std_touch objectcap ...
```

Description

Std_touch executes an `STD_TOUCH` command for each object in its argument list. The purpose of this is to tell the server not to garbage collect the object. (If the capability for an object is lost there is no way to recover the resources it is using other than to garbage collect all objects not accessed within a certain time period.) The exit status of the command is equal to the number of errors encountered.

Diagnostics

Two types of error are possible:

1. Errors while looking up the capability of the object in the directory server.
2. Errors during the actual touch operation itself (such as insufficient permission).

The error messages for the three different types are clear and include the explanation of the error status.

Example

```
std_touch /home/myfile
```

will tell the Bullet Server that */home/myfile* should not be garbage collected since someone still has an interest in it.

See Also

std_age(A), std_copy(U), std_destroy(U), std_info(U), std_restrict(U).

Name

strings – print all the strings in a binary file

Synopsis

`strings [-] [-o] [-`*n*`] file ...`

Description

Strings looks for sequences of ASCII characters followed by a zero byte. These are usually strings. This program is typically used to help identify unknown binary programs.

Options

– search the whole file, not just the data segment.

–o print octal offset of each string.

–*n* Do not show strings less then *n* characters long (default = 4).

Examples

```
strings -5 a.out
```

Print the strings longer than 4 characters in *a.out*.

```
strings - /bin/sh/pd.i80386
```

Search the entire file */bin/sh/pd.i80386* (text and data).

Name

strip – remove the symbol table from executables

Synopsis

```
strip [file] ...
```

Description

Strip removes the symbol table from the executables given as argument. This can be used to reduce disk usage. When no argument is given, the file *a.out* will be stripped by default.

Note that after stripping a binary, it will be virtually useless for debugging purposes.

Example

```
strip foo bar
```

Strips files *foo* and *bar*.

Name

stun – stun (paralyze, possibly kill) a process

Synopsis

```
stun [-d dumpfile] [-m host] [-s stun-code] [-p] [-c]
     [-v] process ...
```

Description

This program sends a ‘‘stun’’ request to the specified processes. The effect of a stun is described in detail in *signals*(L). In short, the normal result of a stun is that the process server in the kernel where the process is running stops the process, generates a process descriptor for the current state of the process and sends it to the owner of the process. If there is no owner or if the owner does not care, the process is destroyed. Usually the purpose of a stun is to destroy a process although it is possible to just take a snapshot of the process state using the **–c** option (described below) and then resume the process.

Processes are named by their capability. The capabilities for a user’s processes are normally stored in the directory */dev/proc* with the POSIX process ID as name. For example, */dev/proc/3*. The name of the process to stun can be determined using *aps*(U) or the command `dir -l /dev/proc`. Note that the process */dev/proc/1* is the session server. Stunning it will terminate the current session.

The capability for a process is also stored in the *ps* directory in the kernel on the host where the process is running. For example, */profile/hosts/venus/ps/1*. Under normal circumstances this capability for the process has insufficient rights to stun it. Those with the */super* capability can use a full path name or the **–m** option to stun a process that belongs to someone else.

Options

–d dumpfile
: Specifies that the core file should have the name *dumpfile*. The default it that it is put in */dev/dead/$$*, where $$ is the process id of the process stunned.

–m host
: Causes the *process* argument(s) to be interpreted as process identifiers in the directory */super/hosts/*`host`*/ps*.

–s stun-code
: Set the stun code. Some owners look at the stun code sent along with the stun request. For example, the session server reports the stun code to a process’ parent as the signal that killed it. The default is 3, or `SIGQUIT`.

–c Print a traceback and then continue the process. *Stun* temporarily takes over ownership of the process, prints a traceback using *pdump*(U), and lets the process continue; the original owner is restored but never notified. Note that this option will not work on the session server process but will kill it.

–p Print a traceback: stun temporarily takes over ownership of the process, prints a traceback using *pdump*(U), and then passes the checkpoint on to the original owner.

Note that this option will not work on the session server process but will kill it.

–v Verbose: print comments about what *stun* is doing.

Warning

The use of the **–c** option is not completely transparent to the process being stunned; *trans*, *getreq* and *mu_trylock* calls may fail causing it to report an error and die.

Examples

To kill process number 4:

```
stun /dev/proc/4
```

To stun a process, see a checkpoint, and see what stun is doing:

```
stun -v -p /dev/proc/4
```

For someone with the */super* capability to kill process number 1 on the host *venus* with the signal –1 (a very sure kill):

```
stun -s -1 -m venus 1
```

See Also

aps(U), kill(U), pdump(U).

Name

sty – show or change tty settings

Synopsis

```
sty [tty cap] [all]
sty [+-]help
sty [tty cap [reset | size] [[+-]flag] ... [[+-]char] ... [char value] ...
```

Description

A tty controls the way the keystrokes typed at the keyboard to a process are interpreted, and the way read and write transactions of the process are handled. The tty interface keeps flags and characters, which can be displayed and set by *sty*. The flags and characters that *sty* understands are defined by POSIX 1003.1. For each flag and character, *sty* knows a default value which is suitable for use on Amoeba. Without any options *sty* prints the current tty settings, insofar as they differ from the compiled-in defaults.

Options

all
: This makes *sty* print every flag and character, instead of just the ones that differ from the defaults.

[+-]**help**
: Prints help: **–help** gives a very short synopsis. **help** gives the synopsis and tells what the defaults for the flags and characters are. **+help** lists the meaning of every flag and character.

tty *capability*
: This tells *sty* to use *capability* as the capability for the tty instead of the default `STDIN` from the capability environment.

reset
: This means use the compiled-in defaults instead of the current tty settings.

size
: This prints the size of the tty in rows and columns of characters, if known.

To turn off a flag, pass the option **–***flag*. To turn on a flag, pass the option *flag*. To set a flag to its default, pass the option **+***flag*.

To disable a ttychar, pass the option **–***charname*. To set a ttychar, pass the option *charname charvalue*, where *charvalue* is of the form *0xhexvalue*, *^character* or *character*. To set a ttychar to its default, pass the option **+***charname*. (Note that if the character ‘‘^’’ is special to the shell, it needs to be quoted or preceded by a backslash.)

The output of *sty* is usable as arguments to *sty* itself, so it is possible to save tty settings in a readable form with

```
sty > Nice_Settings
```

and then later restore them with

```
sty `cat Nice_Settings`
```

For a complete list of characters and flags, consult the POSIX standard, or type

```
sty +help
```

Environment

If no tty option is passed, *sty* uses the capability `STDIN` as the tty capability.

Examples

Three ways to set the ttychar ‘‘eof’’ to CTRL−D are shown:

```
sty +eof
```

works because CTRL−D is the default for this character.

```
sty eof 0x4
```

works because 0x4 happens to be CTRL−D.

```
sty eof '^d'
```

is the common way to set eof to CTRL−D.

To turn off the possibility to signal eof altogether, type:

```
sty -eof
```

To display the complete tty configuration, type:

```
sty all
```

If all settings have their default values, this produces:

```
brkint icrnl -ignbrk -igncr -ignpar -inlcr -inpck -istrip ixoff ixon
-parmrk opost -clocal cread -cstopb -hupcl -parenb -parenodd echo echoe
echok -echol icanon -iexten isig -noflsh -tostop eof '^D' -eol erase '^H'
intr '^C' kill '^X' quit '^´ -susp start '^Q' stop '^S' min 1 time 0
baud 9600
```

See Also

POSIX Standard 1003.1-1988

ax(U), tios_getattr(L), tios_setattr(L), tios_getwsize(L).

Name

sum – compute the checksum and block count of a file

Synopsis

```
sum file ...
```

Description

Sum computes the checksum of one or more files. It prints the checksum and the number of blocks in the file. The block size is 512 bytes. It is most often used to see if a file copied from another machine has been correctly received. This works best when both machines use the same checksum algorithm.

Examples

```
sum /home/xyz
```

might print

```
38349   161
```

Name

tail – print the last few lines of a file

Synopsis

```
tail [-n] [file]
```

Description

The last few lines of one file are printed. The default count is 10 lines. The default file is *stdin*. If more than one *file* argument is given only the first is used. The rest are ignored.

Options

–n Print the last *n* lines of the input. The default is 10.

Examples

```
tail -6
```

Print last 6 lines of *stdin*.

```
tail -1 file1 file2
```

Print last line of *file1*. The second file, *file2*, is ignored.

See Also

head(U).

Name

tape – a general purpose utility for manipulating tapes.

Synopsis

```
tape [-f tape_cap] subcommand
```
subcommands:
```
    read [-r record_size] [-c count] [-v] [-b buffer_count]
    write [-r record_size] [-c count] [-v] [-b buffer_count]
    erase
    fskip [-c file_skip_count]
    rskip [-c record_skip_count]
    load
    unload
    rewind [-w]
    status
```

Description

Tape is a single utility for sending commands to the tape server (see *tape*(A)). The possible tape commands and options are described below.

Options

–f *tape_cap*

This option forces the tape unit specified by *tape_cap* to be used. If this option is not present it uses the capability specified by the string environment variable TAPE, or if this is not present the tape unit specified by the variable DEF_TAPESVR in the include file *ampolicy.h* (typically */profile/cap/tapesvr/default*.)

Subcommands

There are several subcommands with various options and they are described below. The **read** and **write** commands accept several parameters. The default tape record size is 512 bytes but the **–r** option can be used to select another record size. The number of records to be read or written is specified with the **–c** option. The default count is infinity, which will cause the command to read the entire tape or until an EOF marker is encountered. The **–v** option switches on verbose mode. This causes lots of extra detail to be printed. The **–b** option specifies the number of buffers to use for buffering the tape I/O. This is useful when trying to get the tape to stream.

read [-r record_size] [-c count] [-v] [-b buffer_count]

The **read** subcommand extracts data from the specified tape and then writes it to *stdout*.

write [-r record_size] [-c count] [-v] [-b buffer_count]

The **write** subcommand reads data from *stdin* and writes it to the tape.

erase

This command will erase the data from the current position to the end of the tape.

fskip [-c file_skip_count]

This command causes the tape to wind forward until the start of the next file on the tape. It will skip *file_skip_count* files if the **–c** option is given. If there are no more files on the tape it winds forward until the end of the tape.

rskip [-c record_skip_count]

This command causes the tape to wind forward until the start of the next record on the tape. It will skip *record_skip_count* records if the **–c** option is given. If there are no more records on the tape it winds forward until the end of the tape. Some tape controllers skip the EOF tape record as well. This makes the tape file position invalid.

load

On tapes with automatic loading this command will load the tape mounted in the unit. For all tapes it then positions the tape at the start of medium.

unload

This brings the tape to the point where it can be removed from the drive. Some drives will also eject the tape.

rewind [-w]

This command rewinds the tape to the start of medium. If the **–w** flag is given, the command will not return until the tape is rewound. Otherwise it returns immediately without waiting for the command to complete.

status

This command prints the current status of the tape unit. This usually includes the type of the tape unit and the current record and file position.

Diagnostics

In general the error messages suggest what has gone wrong, how many records have been read or written, where relevant, and the error code returned by the tape unit.

Environment Variables

TAPE is a string environment variable which will override the default system tape unit defined in the file *ampolicy.h*. It will not override the **–f** option, if present.

Example

To read the current file from the default tape unit and store it in the file *foo* use the command:

```
tape read > foo
```

See Also

tape(A), tape(L).

Name

tar – tape (or other media) file archiver

Synopsis

```
tar [BcdDhikmopRstvxzZ] [-b N] [-f file] [-T file] [file|regexp] ...
```

Description

Tar provides a way to store many files into a single archive, which can be kept in another UNIX file, stored on an I/O device such as tape, floppy, cartridge, or disk, or piped to another program. It is useful for making backup copies, or for packaging up a set of files to move them to another system.

Tar has existed since Version 7 UNIX with very little change. It has been proposed as the standard format for interchange of files among systems that conform to the IEEE P1003 ‘‘Portable Operating System’’ standard.

When reading an archive, this version of *tar* continues after finding an error. Previous versions required the **i** option to ignore checksum errors.

Note that *tar* does not go into infinite loops in the presence of cycles in the directory graph. Each file will be archived once. However, it may not always restore links correctly. *Starch*(U) attempts to handle links properly.

Options

Tar options can be specified in either of two ways. The usual UNIX conventions can be used: each option is preceded by ‘–’; arguments directly follow each option; multiple options can be combined behind one ‘–’ as long as they take no arguments. For compatibility with the UNIX *tar* program, the options may also be specified as ‘‘keyletters,’’ wherein all the option letters occur in the first argument to *tar*, with no ‘–’, and their arguments, if any, occur in the second, third, ... arguments. Examples:

Normal: `tar -f arcname -cv file1 file2`

Old: `tar cvf tarfile file1 file2`

At least one of the **–c**, **–t**, **–d**, or **–x** options must be included. All others are optional.

Files to be operated upon are specified by a list of file names, which follows the option specifications (or can be read from a file by the **–T** option). Specifying a directory name causes that directory and all the files it contains to be (recursively) processed. If a full path name is specified when creating an archive, it will be written to the archive without the initial ‘‘/’’, to allow the files to be later read into a different place than where they were dumped from, and a warning will be printed. If files are extracted from an archive which contains full path names, they will be extracted relative to the current directory and a warning message will be printed.

When extracting or listing files, the ‘‘file names’’ are treated as regular expressions, using mostly the same syntax as the shell. The shell actually matches each substring between ‘‘/’’s separately, while *tar* matches the entire string at once, so some anomalies will occur; e.g., ‘‘*’’ or ‘‘?’’ can match a ‘‘/’’. To specify a regular expression as an argument to *tar*, quote it

so the shell will not expand it.

–b *N* Specify a blocking factor for the archive. The block size will be *N* x 512 bytes. Larger blocks typically run faster and fit more data on a tape. The default blocking factor is set when *tar* is compiled, and is typically 20. There is no limit to the maximum block size, as long as enough memory can be allocated for it, and as long as the device containing the archive can read or write that block size.

–B When reading an archive, reblock it as it is read. Normally, *tar* reads each block with a single *read()* call. This does not work when reading from a pipe; *read()* only gives as much data as has arrived at the moment. With this option, it will do multiple reads to fill out to a record boundary, rather than reporting an error. This option is default when reading an archive from standard input.

–c Create an archive from a list of files.

–d Diff an archive against the files in the file system. Reports differences in file size, mode, uid, gid, and contents. If a file exists on the tape, but not in the file system, that is reported. This option needs further work to be really useful.

–D When creating an archive, only dump each directory; do not dump all the files inside the directory. In conjunction with *find*(U), this is useful in creating incremental dumps for archival backups.

–f file Specify the filename of the archive. If the specified filename is ‘‘–’’, the archive is read from the standard input or written to the standard output. If the **–f** option is not used, and the environment variable TAPE exists, its value will be used; otherwise, a default archive name (which was picked when tar was compiled) is used. The default is normally set to the ‘‘first’’ tape drive or other transportable I/O medium on the system.

–h When creating an archive, if a symbolic link is encountered, dump the file or directory to which it points, rather than dumping it as a symbolic link.

–i When reading an archive, ignore blocks of zeros in the archive. Normally a block of zeros indicates the end of the archive, but in a damaged archive, or one which was created by appending several archives, this option allows *tar* to continue. It is not on by default because there is garbage written after the zeroed blocks by the UNIX *tar* program. Note that with this option set, *tar* will read all the way to the end of the file, eliminating problems with multi-file tapes.

–k When extracting files from an archive, keep existing files, rather than overwriting them with the version from the archive.

–m When extracting files from an archive, set each file’s modified time-stamp to the current time, rather than extracting each file’s modified time-stamp from the archive.

–o When creating an archive, write an old format archive, which does not include information about directories, pipes, fifos, contiguous files, or device files, and specifies file ownership by uid’s and gid’s rather than by user names and group names. In most cases, a ‘‘new’’ format archive can be read by an ‘‘old’’ tar program without serious trouble, so this option should seldom be needed.

–p When extracting files from an archive, restore them to the same permissions that they had in the archive. If **–p** is not specified, the current umask limits the

permissions of the extracted files.

–R With each message that *tar* produces, print the record number within the archive where the message occurred. This option is especially useful when reading damaged archives, since it helps to pinpoint the damaged section.

–s When specifying a list of filenames to be listed or extracted from an archive, the **–s** flag specifies that the list is sorted into the same order as the tape. This allows a large list to be used, even on small machines, because the entire list need not be read into memory at once. Such a sorted list can easily be created by running `tar –t` on the archive and editing its output.

–t List a table of contents of an existing archive. If file names are specified, just list files matching the specified names. The listing appears on the standard output.

–T file Rather than specifying file names or regular expressions as arguments to the *tar* command, this option specifies that they should be read from the file *F*, one per line. If the file name specified is ‘‘–’’, the list is read from the standard input. This option, in conjunction with the **–s** option, allows an arbitrarily large list of files to be processed, and allows the list to be piped to *tar*.

–v Be verbose about the files that are being processed or listed. Normally, archive creation, file extraction, and differencing are silent, and archive listing just gives file names. The **–v** option causes an ‘‘ls –l’’–like listing to be produced. The output from **–v** appears on the standard output except when creating an archive (since the new archive might be on standard output), where it goes to the standard error output.

–x Extract files from an existing archive. If file names are specified, just extract files matching the specified names, otherwise extract all the files in the archive.

–z or **–Z**

The archive is compressed as it is written, or decompressed as it is read, using the *compress*(U) program. This option works on I/O devices as well as on disk files; data to or from such devices is reblocked using the *dd*(U) command to enforce the specified (or default) block size. The default compression parameters are used; to override them, avoid the **–z** option and compress it afterwards.

Warnings

The **r**, **u**, **w**, **X**, **l**, **F**, **C**, and *digit* options of UNIX *tar* are not supported.

Multiple-tape (or floppy) archives should be supported, but so far no clean way has been implemented.

A bug in the Bourne Shell usually causes an extra newline to be written to the standard error when using compressed archives.

A bug in *dd*(U) prevents turning off the ‘‘x+y records in/out’’ messages on the standard error when *dd* is used to reblock or transport an archive.

See Also

aal(U), compress(U), dd(U), find(U), shar(U), starch(U).

Name

tee – divert *stdin* to a file

Synopsis

```
tee [-ai] file ...
```

Description

Tee copies its standard input to its standard output. It also makes copies on all the files listed as arguments.

Options

–a Append to the files, rather than overwriting them.

–i Ignore interrupts.

Examples

```
cat file1 file2 | tee xxx
```

Save the concatenation of the contents of the two files *file1* and *file2* in *xxx* and also display it on the standard output.

```
pr file | tee x | lpr
```

Save the output of *pr* on *x* while also sending it to the program *lpr*.

Name

termcap – print the current termcap entry

Synopsis

```
termcap [type]
```

Description

Termcap reads the file */etc/termcap* to obtain the entry corresponding to the terminal type supplied as the argument *type*. If none is given, the current value of the environment variable `TERM` is used. It then prints out all the parameters that apply.

Example

```
termcap
```

Print the termcap entry for the current terminal which might look like

```
TERM = sun

End of line wraps to next line   (am)
Ctrl/H performs a backspace      (bs)
Number of columns                (co) = 80
Number of lines                  (li) = 34
Clear to end of line             (ce) = ^[[K
Clear to end of screen           (cd) = ^[[J
Clear the whole screen           (cl) = ^L
Start "stand out" mode           (so) = ^[[7m
End "stand out" mode             (se) = ^[[m
Start underscore mode            (us) = ^[[4m
End underscore mode              (ue) = ^[[m
Start bold mode                  (md) = ^[[1m
Start reverse mode               (mr) = ^[[7m
Return to normal mode            (me) = ^[[m
Cursor motion                    (cm) = ^[[%i%d;%dH
Up one line                      (up) = ^[[A
Right one space                  (nd) = ^[[C
Generated by "UP"                (ku) = ^[[A
Generated by "DOWN"              (kd) = ^[[B
Generated by "LEFT"              (kl) = ^[[D
Generated by "RIGHT"             (kr) = ^[[C
Generated by "PGUP"              (k1) = ^[[224z
Generated by "PGDN"              (k2) = ^[[225z
Generated by numeric "+"         (k3) = ^[[226z
Generated by numeric "-"         (k4) = ^[[227z
Generated by numeric "5"         (k5) = ^[[228z
```

Name

test – test for a condition

Synopsis

```
test expr
[ expr ]
```

Description

Test checks to see if objects exist, are readable, etc. and returns an exit status of zero if true and nonzero if false. The legal operators are

–r object true if the file is readable.

–w object true if the file is writable.

–x file true if the file is a directory and readable or an executable file. A file is executable if it begins with a process descriptor or begins with the string ‘‘#!’’.

–f file true if the object is an Amoeba file (and not a directory or some other object type).

–d object true if the object is a directory.

–s file true if the file exists and has a size > 0.

–t fd true if file descriptor *fd* (default 1) is a terminal.

–z s true if the string *s* has zero length.

–n s true if the string *s* has nonzero length.

s1 = *s2* true if the strings *s1* and *s2* are identical.

s1 **!=** *s2* true if the strings *s1* and *s2* are different.

m **–eq** *n* true if the integers *m* and *n* are numerically equal.

The operators **–gt**, **–ge**, **–ne**, **–le** and **–lt** may be used as well in place of **–eq**. These operands may be combined with **–a** (Boolean and), **–o** (Boolean or), **!** (negation). The priority of **–a** is higher than that of **–o**. Parentheses are permitted, but must be escaped to keep the shell from trying to interpret them.

Warning

Test is built-in in some shells and may implement different semantics.

Example

```
test -r file
```

See if *file* is readable.

```
[ -r file ]
```

The same but with the other syntax.

Name

time – time how long a command takes to execute

Synopsis

```
time command
```

Description

The command is executed and the real time, user time, and system time (in hours, minutes, and seconds) are printed.

Warning

Currently there is no way of measuring user and system time so they are printed as zero.

Examples

```
time a.out
```

Reports how long *a.out* takes. The time report look like this:

```
10.64s real    0.00s user    0.00s system
```

Name

tob – copy a UNIX file to an Amoeba Bullet Server

Synopsis

```
tob [-n] unix-name amoeba-name [fileserver]
```

Description

Tob is a command which runs only under UNIX with an Amoeba driver. It reads the file *unix-name* from the UNIX system and stores it on an Amoeba Bullet Server under the name *amoeba-name* in the Amoeba directory service. This name must not exist already.

The optional third parameter specifies the path name in the Amoeba directory server for the capability of a particular Bullet Server to use. If the third parameter is omitted then the default Bullet Server is used.

Options

The **–n** option requests that the capabilities for the Bullet file and optional fileserver not be stored or looked up in the Amoeba directory server. Instead they should be stored or looked up in UNIX files with the specified name(s). This feature is for bootstrapping a system when no directory server is available.

Example

```
tob /etc/termcap /home/mytermcap
```

will copy the UNIX file */etc/termcap* to the default Bullet Server and store under the name */home/mytermcap* under Amoeba.

See Also

bullet(A), fromb(U).

Name

touch – update the time on the directory entry of an object

Synopsis

```
touch [-c] pathname ...
```

Description

Touch sets the creation time on the directory entry for an object to the current time. Since the time is stored per directory entry and not per object it will not update other directory entries which point to the same object. If the directory entry does not exist then an empty file will be created and registered under the *pathname* specified.

Options

–c Do not create a file and directory entry if the directory entry does not exist.

Example

```
touch /home/foo
```

will set the creation time to the current time on the directory entry */home/foo*.

Name

tr – translate character codes

Synopsis

```
tr [-cds] [string1] [string2]
```

Description

Tr performs simple character translation. When no flag is specified, each character in *string1* is mapped onto the corresponding character in *string2*.

A string may consist of one or more of

range e.g.: [a-z] or [0-9]

escape e.g.: \012 (meaning newline) or \\ (‘‘\’’ itself)

other e.g.: A or (or 1

Options

–c Complement the set of characters in string1.

–d Delete all characters specified in string1.

–s Squeeze all runs of characters in string1 to one character.

Examples

```
tr ’[a-z][A-Z]’ ’[A-Z][a-z]’ < x > y
```

Convert upper case to lower case, and vice versa, reading the input from the file *x* and writing it to the file *y*.

```
tr -d ’0123456789’ < f1 > f2
```

Write on the file *f2* the contents of *f1* but with all the digits deleted.

Name

treecmp – recursively compare two directory trees

Synopsis

```
treecmp [-v] dir1 dir2
```

Description

This program recursively compares the two directory trees *dir1* and *dir2* and reports any differences. If the command line arguments refer to files rather than directories then it will report any differences between the two files, although *diff* or *cmp* may be better suited to this purpose. It can be used, for example, when a project consists of a large number of files and directories. When a new release (i.e., a new tree) has been prepared, the old and new tree can be compared to give a list of what has changed.

The algorithm used is that the first tree is recursively descended and for each file or directory found, the corresponding one in the other tree checked. The two arguments are not completely symmetric because the first tree is descended, not the second one, but reversing the arguments will still detect all the differences, only they will be printed in a different order.

Options

–v Verbose mode. It prints the names of directories as it enters them.

Warnings

This program may not handle cycles in the directory graph. It is only likely to work properly when applied to directory (sub)graphs with a tree structure.

It will also be confused by objects other than directories and files. Objects of unknown type will be reported with the message

```
Unknown file type nnn
```

where *nnn* is the octal representation of the files type.

See Also

cdiff(U), cmp(U), diff(U).

Name

true – exit with the value true

Synopsis

```
true
```

Description

This command returns the value true (i.e. 0). It is used for shell programming.

Example

```
while true
do
        ls -l
done
```

List the directory until interrupted.

See Also

false(U).

Name

tset – set the TERM string environment variable

Synopsis

```
tset [device]
```

Description

Tset is used almost exclusively to set the shell variable `TERM` from inside profiles. If an argument is supplied, that is used as the value of `TERM`. Otherwise it looks in */etc/ttytype*.

Example

```
eval `tset`
```

Sets the value of `TERM`.

Name

tsort – topological sort

Synopsis

```
tsort file
```

Description

Tsort accepts a file of lines containing ordered pairs and builds a total ordering from the partial orderings.

Name

ttn – a simple telnet implementation

Synopsis

`ttn [-tcp` *tcp_capability*`] host [port]`

Description

Ttn implements the Telnet protocols as described in RFC-854. The implementation is very simple and only a few options are supported. Supported options are the ‘‘echo’’ option (option 1, as described in RFC-857), the ‘‘terminal type’’ option (option 24, as described in RFC-930) and the ‘‘suppress go ahead’’ option (option 3, as described in RFC-858).

The *host* option is the name of the IP host to connect to. This can be a host number (of the form nn.nn.nn.nn) or a host name. The *port* option specifies the destination TCP port. The default is the telnet port (23) but other ports are possible. This can be a number or a name of a service in the */etc/services* file.

Options

–tcp *tcp_capability*
This option gives the path name for the TCP server to be used.

Diagnostics

Ttn writes on *stdout* where it is connecting to. For example,

```
connecting to 121.97.19.3 23
```

where the *121.97.19.3* represents the Internet address of the destination and the *23* is the TCP port. After a successful connect

```
connect succeeded !!!!
```

is written to *stdout*.

Environment Variables

TCP_SERVER
this environment variable overrides the default path for the tcp capability. Note that the command line option has the highest priority, followed by the environment variable and the default has the lowest priority.

Files

/etc/services

Examples

```
ttn ski
```

Telnet to the telnet port of the host *ski*.

```
ttn ski finger
```

Connect to the finger daemon port of the host *ski*.

See Also

ip(A), ip(L).

Name

tty – print the device name of this tty

Synopsis

```
tty [-s]
```

Description

Print the name of the controlling tty.

Options

–s Silent mode: return status only.

Example

```
tty
```

Might print */dev/tty*. This name does not actually appear in the directory graph, but it is a name which is recognized by the POSIX emulation.

Name

unexpand – convert spaces to tabs

Synopsis

```
unexpand [-a] [file ...]
```

Description

Unexpand replaces leading spaces on each line with tabs in each of the named files. If no files are given, *stdin* is used. The output is written to *stdout*.

Options

–a The **–a** flag is used to force all sequences of spaces to be converted, instead of just leading spaces (the default).

Example

```
unexpand oldfile > newfile
```

Convert leading spaces in *oldfile* to tabs and save the result in *newfile*.

See Also

expand(U).

Name

uniq – delete consecutive identical lines in a file

Synopsis

```
uniq [-cdu] [+n] [-n] [input [output]]
```

Description

Uniq examines the file *input* for consecutive lines that are identical. All duplicate entries are deleted, and the file is written to the file *output*.

Options

–c The count of identical copies of each line is written at the start of each output line.

–d Only duplicate lines are written to output. One copy of each line that had duplicates is written.

+n When comparing lines skip the first *n* fields, where a field is defined as a run of characters separated by white space.

–n When comparing lines skip the first *n* spaces. Fields are skipped first.

–u Only unique lines are written to output.

Examples

```
uniq +2 file
```

Ignore first 2 fields when comparing.

```
uniq -d inf outf
```

Write duplicate lines from the file *inf* to *outf*.

See Also

sort(U).

Name

uud – decode a binary file encoded with uue

Synopsis

```
uud [-d] [-n] [-s srcdir/] [-t dstdir/] file
```

Description

Uud decodes a file encoded with *uue*(U) or *uuencode*. The decoded file is given the name that the original file had. The name information is part of the encoded file. Mail headers and other junk before the encoded file are skipped. If *file* is ‘‘-’’ then *stdin* is used.

Options

–d Print debug information.

–n Do not verify checksums.

–s *srcdir/*
Srcdir/ is the name of directory where the *.uue* file is.

–t *dstdir/*
Dstdir/ is the name of directory where the output goes.

Examples

```
uud file.uue
```

Re-create the original file.

```
uud - < file.uue
```

The ‘‘–’’ means use *stdin*.

See Also

uue(U).

Name

uue – encode a binary file to ASCII (e.g., for mailing)

Synopsis

```
uue [-n] file [-]
```

Description

Uuencode is a famous program that converts an arbitrary (usually binary) file to an encoding using only 64 ASCII characters. Uudecode converts it back to the original file. The *uue* and *uud*(U) programs are enhanced versions of these programs, and are compatible with the UNIX ones. The files produced can even be sent successfully over BITNET, which is notorious for mangling files.

It is possible to have *uue* automatically split the encoded file up into chunks small enough to fit into mail messages. The output files then get the suffixes *.uaa*, *.uab*, etc., instead of *.uue*. When *uud* is given *file.uaa* to decode, it automatically includes the subsequent pieces. The encoding takes 3 bytes (24 bits) from the input file and renders it as 4 bytes in the output file.

Options

–n *n* is a decimal integer specifying how many lines to put in each output file.

Examples

```
uue file
```

Encode *file* to *file.uue*.

```
uue file - > x
```

Encode *file* and write the result on *stdout*.

```
uue -800 file
```

Output on *file.uaa*, *file.uab* etc.

See Also

uud(U).

Name

wc – count characters, words, and lines in a file

Synopsis

```
wc [-clw] file ...
```

Description

Wc reads each *file* argument and computes the number of characters, words and lines it contains. A word is delimited by white space (space, tab, or line feed). If no flags are present, all three counts are printed.

Options

–c Print the character count.

–l Print the line count.

–w Print the word count.

The default is to print all three.

Examples

```
wc file1 file2
```

Print all three counts for both files. This might look like:

```
      3       8     50 file1
    254    2001  17344 file2
    257    2009  17394 total
```

```
wc -l file
```

Prints the line count only. It might look like:

```
      3 file
```

Name

which – examine PATH to see which file will be executed

Synopsis

```
which name
```

Description

The PATH environment variable controls the search rules for executables. If a command *a.out* is given, the shell first tries to find an executable file in the working directory. If that fails, it looks in various directories, according to the value of PATH. The *which* command makes the same search, and gives the absolute path of the program that will be chosen, followed by other occurrences of the file name along the path.

Example

```
which a.out
```

Reports the path name of the program *a.out* that will be executed.

Name

who – report who is logged onto the system

Synopsis

```
who
```

Description

Who prints a list of the login names of the people logged in to Amoeba. It determines this by running *aps*(U) and looking for session servers. It extracts some relevant fields from the output of *aps* and prints them.

Example

```
who
```

might print

```
versto    armada00    session -d -a
keie      armada01    session -a
geels     armada01    session -a
phil      armada03    session -a
sater     armada03    session -a
leendert  armada03    -session -a /bin/sh
bahler    armada09    session -a
ast       armada09    session -a
kaashoek  vmesa4      session -a
gregor    vmesc2      session -a
```

See Also

whoami(U).

Name

whoami – print user's login name

Synopsis

```
whoami
```

Description

Whoami tells in which protection domain it is run. This is done by comparing the `ROOT` capability environment variable with the capabilities in */profile/group/users*.

Environment Variables

`ROOT` – the capability searched for, i.e., the capability for ''/''.

Files

/profile/group/users – the directory with users root directories.

See Also

who(U).

Name

X – the set of X utilities that are known to work under Amoeba

Synopsis

```
The list of X utilities that work under Amoeba
```

Description

The various programs delivered with the X windows system should all work under Amoeba. In particular the X servers and the client programs *twm* and *xterm* all work. To use the X utilities add */profile/module/x11/bin* to your PATH environment variable.

If X windows has not been installed, consult your local system administrator.

Name

xargs – execute a command with many arguments

Synopsis

```
xargs command [initial-arguments]
```

Description

Xargs is based on the UNIX System V command of the same name. This version performs only the default function of *xargs*, which is also the simplest function. It combines the *initial-arguments* with the arguments read from standard input and executes the *command* one or more times with those arguments.

Because there is a limit on how much space the arguments of a command may use, if the argument list is too large the system prohibits the command from being executed (in System V, this error is `E2BIG`, *errno* = 7).

To avoid this problem, *xargs* will take arguments which are themselves a command and *its* arguments (options), and then read standard input and apply it to the command with its given arguments. *Xargs* is careful not to exceed the system-imposed limit, which is expected to be greater than the system's stream file buffer size, *BUFSIZ* in *stdio.h*. It continues to execute the command with the read-in arguments until it reaches end-of-file.

Name

xreinit – reinitialize an X server

Synopsis

```
xreinit display
```

Description

Xreinit causes the specified X server to reinitialize. Among other things, this causes all windows to be destroyed and all connections to be closed.

Warning

This call is specific to the Amoeba X server, so it will not work for X servers running on UNIX.

Example

The following line in your *.profile*

```
trap 'xreinit $DISPLAY' 0
```

will close all windows upon logout, leaving the X server ready for a new session.

See Also

X(U), xshutdown(U).

Name

xshutdown – perform orderly shutdown of an X server

Synopsis

```
xshutdown display
```

Description

Xshutdown performs an orderly shutdown of an X server. All connections are closed, and the server terminates. It is useful when shutting down a workstation.

Warning

This call is specific to the Amoeba X server, so it will not work for X servers running on UNIX.

Example

```
xshutdown myhost:0
```

terminates the X server on *myhost* cleanly.

See Also

X(U), xreinit(U).

Name

yacc – an LALR(1) parser generator

Synopsis

```
yacc [ -dltv ] [ -b prefix ] filename
```

Description

Yacc reads the grammar specification in the file *filename* and generates an LR(1) parser for it. The parsers consist of a set of LALR(1) parsing tables and a driver routine written in the C programming language. *Yacc* normally writes the parse tables and the driver routine to the file *y.tab.c*.

Options

The following options are available:

–b prefix This option changes the prefix prepended to the output file names to the string denoted by *prefix*. The default prefix is the character *y*.

–d This option causes the header file *y.tab.h* to be written.

–l If the **–l** option is not specified, *yacc* will insert #line directives in the generated code. The #line directives let the C compiler relate errors in the generated code to the user's original code. If the **–l** option is specified, *yacc* will not insert the #line directives. #line directives specified by the user will be retained.

–t This option will change the preprocessor directives generated by *yacc* so that debugging statements will be incorporated into the compiled code.

–v This option causes a human-readable description of the generated parser to be written to the file *y.output*.

If the environment variable `TMPDIR` is set, the string denoted by `TMPDIR` will be used as the name of the directory where the temporary files are created.

Tables

There is a program *:yyfix* that extracts tables from *yacc*-generated files. The program takes the names of the tables as its command-line arguments. The names of the tables generated by this version of *yacc* are *yylhs*, *yylen*, *yydefred*, *yydgoto*, *yysindex*, *yyrindex*, *yygindex*, *yytable*, and *yycheck*. Two additional tables, *yyname* and *yyrule,* are created if `YYDEBUG` is defined and nonzero.

Files

y.tab.c
y.tab.h
y.output
/tmp/yacc.aXXXXXX
/tmp/yacc.tXXXXXX
/tmp/yacc.uXXXXXX

Diagnostics

If there are rules that are never reduced, the number of such rules is reported on standard error. If there are any LALR(1) conflicts, the number of conflicts is reported on standard error.

Name

yap – yet another pager

Synopsis

```
yap [-num] [-cnqu] [+command] [filename ...]
```

Description

Yap is a program that allows the user to examine a continuous text, one screen full at a time on a video display terminal. It does so by pausing after each screen full, waiting for the user to type a command. The commands are described below.

If no file name is given *yap* reads from standard input.

*Yap'*s main feature is that it can page both forwards and backwards, even when reading from standard input.

Options

–num An integer which is the size (in lines) of a page (the initial *page-size*).

–c Normally, *yap* will display each page by beginning at the top of the screen and erasing each line just before it displays on it. If the terminal cannot erase a line, *yap* will clear the screen before it displays a page.
This avoids scrolling the screen, making it easier to read while *yap* is writing. The **–c** option causes *yap* to scroll the screen instead of beginning at the top of the screen. This is also done if the terminal can neither erase a line or clear the screen.

–n Normally, *yap* recognizes escape sequences for stand-out mode or underlining mode in the input, and knows how much space these escape sequences will occupy on the screen, so that *yap* will not fold lines erroneously. The **–n** option suppresses this pattern matching.

–q This option will cause *yap* to exit only on the ‘‘quit’’ command.

–u Normally, *yap* handles underlining such as produced by nroff in a manner appropriate to the particular terminal: if the terminal can perform underlining well (ie., the escape sequences for underlining do not occupy space on the screen), *yap* will underline underlined information in the input. The **–u** option suppresses this underlining.

+command This is taken to be an initial command to *yap*. (See the description of the commands below.)

Environment Variables

Yap uses the *termcap* database to determine the terminal capabilities and the default *page-size*. It examines the `TERM` environment variable to identify the terminal type. If `TERM` is not set, it defaults to *dumb*. *Yap* also examines the `TERMCAP` environment variable to locate the *termcap* database. If `TERMCAP` is not set, it defaults to */etc/termcap*.

Yap looks in the `YAP` environment variable to pre-set flags. For instance, if the **–c** mode of

operation is preferred, just set the YAP environment variable to **–c**.

The commands of *yap* can be bound to sequences of key-strokes. The environment variable YAPKEYS may contain the bindings in the form of a list of colon-separated 'name=sequence' pairs. The *name* is a short mnemonic for the command, the *sequence* is the sequence of key-strokes to be typed to invoke the command. This sequence may contain a ^X escape, which means CTRL–X, and a \X escape, which means X. The latter can be used to get the characters '^', '\' and ':' in the sequence. There are two keymaps available, the default one and a user-defined one. The *change keymap* command is used to switch between them.

Commands

The *yap* commands are described below. The mnemonics for the commands are given in parentheses. The default key sequences (if any) are given after the mnemonic. Every command takes an optional integer argument, which may be typed before the command. Some commands just ignore it. The integer argument is referred to as *i*. Usually, if *i* is not given, it defaults to 1.

visit previous file (*bf*) **P**
: Visit the *i*-th previous file given on the command line.

scroll one line up or go to line (*bl*) **^K** *or* **k**
: If *i* is not given, scroll one line up. Otherwise, *i* will be interpreted as a line number. A page starting with the line indicated will then be displayed.

bottom (*bot*) **l** *or* **$**
: Go to the last line of the input.

display previous page (*bp*) **–**
: Display the previous page, consisting of *i* lines, (or *page-size* lines if no argument is given).

display previous page and set pagesize (*bps*) **Z**
: Display the previous page, consisting of *i* lines, (or *page-size* lines if no argument is given). If *i* is given, the *page-size* is set to *i*.

scroll up (*bs*) **^B**
: Scroll up *i* lines (or *scroll-size* lines if *i* is not given. The initial *scroll-size* is 11).

search backwards for pattern (*bse*) **?**
: Search backwards for the *i*-th occurrence of a regular expression which will be prompted for. If there are less than *i* occurrences of the expression, the position in the file remains unchanged. Otherwise, a page is displayed, starting two lines before the place where the expression was found. The user's erase and kill characters may be used to edit the expression. Erasing back past the first character cancels the search command.

skip lines backwards (*bsl*) **S**
: Skip *i* lines backwards and display a page.

skip pages backwards (*bsp*) **F**
: Skip *i* pages backwards and display a page.

scroll up and set scrollsize (*bss*) **b**
: Scroll up *i* lines (or *scroll-size* lines if *i* is not given. If *i* is given, the *scroll-size* is

set to *i*.

change key map (*chm*) **^[**
: Change from the current key map to the other (if there is one).

exchange current page and mark (*exg*) **x**
: Set the mark to the current page, and display the previously marked page.

visit next file (*ff*) **N**
: Visit the *i*-th next file given in the command line.

scroll one line down or go to line (*fl*) **^J** *or* **^M** *or* **j**
: If *i* is not given, scroll one line down. Otherwise, *i* will be interpreted as a line number. A page starting with the line indicated will then be displayed.

display next page (*fp*) **<space>**
: Display the next page, consisting of *i* lines, (or *page-size* lines if no argument is given).

display next page and set pagesize (*fps*) **z**
: Display the next page, consisting of *i* lines, (or *page-size* lines if no argument is given). If *i* is given, the *page-size* is set to *i*.

scroll down (*fs*) **^D**
: Scroll down *i* lines (or *scroll-size* lines if no argument is given).

search forwards for pattern (*fse*) **/**
: Search forwards for the *i*-th match of a regular expression which will be prompted for. If there are less than *i* matches of the expression, the position in the file remains unchanged. Otherwise, a page is displayed, starting two lines before the place where the match was found. The user's erase and kill characters may be used to edit the expression. Erasing back past the first character cancels the search command.
: Note: Some systems do not have *regexp*(L). On those systems, searches are still supported, but regular expressions are not.

skip lines forwards (*fsl*) **s**
: Skip *i* lines and display a page.

skip pages forwards (*fsp*) **f**
: Skip *i* pages and display a page.

scroll down and set scrollsize (*fss*) **d**
: Scroll down *i* lines (or *scroll-size* lines if *i* is not given). If *i* is given, the *scroll-size* is set to *i*.

help (*hlp*) **h**
: Give a short description of all commands that are bound to a key sequence.

set a mark (*mar*) **m**
: Set a mark on the current page.

repeat last search (*nse*) **n**
: Search for the *i*-th occurrence of the last regular expression entered, in the direction of the last search.

repeat last search in other direction (*nsr*) **r**
: Search for the *i*-th occurrence of the last regular expression entered, but in the other direction.

quit (*qui*) **Q** *or* **q**
: Exit from *yap*.

redraw (*red*) **^L**
: Redraw the current page.

repeat (*rep*) **.**
: Repeat the last command. This does not always make sense, so not all commands can be repeated.

shell escape (*shl*) **!**
: Invoke the shell with a command that will be prompted for. In the command, the characters '%' and '!' are replaced with the current file name and the previous shell command respectively. The sequences '\%' and '\!' are replaced by '%' and '!' respectively. The user's erase and kill characters can be used to edit the command. Erasing back past the first character cancels the command.

pipe to shell command (*pip*) **|**
: Pipe the current input file into a shell command that will be prompted for. The comments given in the description of the shell escape command apply here too.

go to mark (*tom*) **'**
: Display the marked page.

top (*top*) **^^**
: Display a page starting with the first line of the input.

visit file (*vis*) **e**
: Visit a new file. The file name will be prompted for. If return is typed the current file is revisited.

write input to a file (*wrf*) **w**
: Write the input to a file, whose name will be prompted for.

The commands take effect immediately, ie. it is not necessary to type a carriage return. Up to the time when the command sequence itself is given, the user may give an interrupt to cancel the command being formed.

Diagnostics

One hopes these are self-explanatory.

Files

/etc/termcap – terminal capabilities data base.

Warnings

Yap will treat the terminal as very stupid if it has no way of placing the cursor on the home position, or can neither erase a line or insert one.

In lines longer than about 2000 characters, a linefeed is silently inserted.

The percentage of the file read, given in the prompt when *yap* reads from a file (and knows it), is not always very accurate.

Example

```
yap myfile
```

will display the text in *myfile* on the screen.

See Also

termcap(L), regexp(L).

Name

yes – repeatedly write a string (by default ‘‘y’’) to *stdout*

Synopsis

```
yes [-n] [string]
```

Description

Yes repeatedly writes *string* to *stdout*, followed by a new-line. If no *string* is entered on the command line, ‘‘y’’ is taken to be the default string. The program can be terminated by typing an interrupt character.

Options

–n Suppress the new-line at the end of the string.

Examples

```
yes -n bla
```

Fill every line with ‘‘blablabla...’’.

```
yes | rm -i *.c
```

Remove all the files ending with *.c* without waiting for a confirmation from the user, even if he does not have write permission on some or all files. Obviously the user needs write permission on the directory.

8 Index

The italic references in the index are to programs or routines described in this manual.

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z