PAGER — Environmental Variable

Specify Output Filter
PAGER="*command options*"

The environmental variable **PAGER** directs programs such as **msgs**, **mail** and others to "pipe" their output into the *command* specified as the value of **PAGER**. For example, the following sets up **/bin/scat** as the desired output filter and passes a command line option to it to specify that the output screen has 20 lines.

```
export PAGER="exec /bin/scat -l20"
```

See Also

scat, environmental variables, mail, more, msgs

param.h — Header File

Define machine-specific parameters
#include <sys/param.h>

param.h defines machine-specific parameters. These parameters set limits on the operation of the COHERENT system; e.g., the number of files that can be open at any one time.

See Also

header files

Notes

This header file is obsolete, and will be dropped from a future release of COHERENT. Its use is strongly discouraged.

passwd — Command

Set/change login password
passwd [*user***]**

passwd sets or changes the password for the specified *user*. If *user* is not specified, **passwd** changes the password of the caller.

passwd requests that the old password (if any) be typed, to ensure the caller is who he claims to be. Next it requests a new password, and then requests it again in case of typing errors. **passwd** requests a longer password if the one given is too brief or too simple.

Files

/etc/shadow — Encrypted passwords

See Also

commands, crypt(), login

Notes

One good way to construct a password is to concatenate two common words plus a punctuation mark. For example, "dog~collar" or "hamlet&horatio" are passwords that are both easy to remember and difficult to guess.

passwd — System Administration

Define system users

The file **/etc/passwd** holds information about each user who has permission to use the COHERENT system. This information is read by the commands **login** and **passwd** whenever a user attempts to log in, to ensure that that user is really himself and not an impostor.

/etc/passwd holds one record for each user; each record, in turn, consists of seven colon-separated fields, as follows:

```
name:password:user_id:group_id:comments:home_dir:shell
```

name is the user's login name.

password is his encrypted password. If this field holds a single asterisk '*', then the program **login** reads his password out of the file **/etc/shadow**.

user_id is a unique number that is also used to identify the user. *group_id* identifies the group to which the user belongs, if any.

comments holds miscellaneous data, such as names, telephone numbers, or office numbers.

home_dir gives the user's home directory.

Finally, *shell* gives the program that is first executed when the user logs on; in most instances, this is an interactive shell (default, **/bin/sh**).

If you wish, you can set additional passwords to control users who attempt to log into your system remotely (that is, via a modem). You can set a different remote-access password for each group of users, based on the program invoked when they log in; for example, you can set one password for the users who log in and invoke **uucico**, and another for the users who log in and invoke the interactive shells **ksh** or **sh**. For details on how to do this, see the Lexicon entries **d_passwd** and **dialups**.

When a user creates a file, that file is "owned" by him. For example, whenever user **joe** create a file, that file is "owned" by **joe**; and **joe** has user-level permissions on that file. The superuser **root** can use the command **chown** to change the ownership of a file from one user to another. For details on this command, see its entry in the Lexicon.

See Also

Administering COHERENT, chown, passwd [command]

Notes

/etc/passwd can be read by anyone: if access to it were refused to a user, he could not log on. Thus, the passwords encrypted within it can be read and copied by anyone, and so may be vulnerable to brute-force decryption. For this reason, close attention should be paid to passwords: they should not be common words or names, preferably mix cases or use unique spellings, and be at least six characters long.

paste — Command

Merge lines of files
paste [-s] [-d *list***]** *file ...*

paste merges corresponding lines from multiple input files. By default, **paste** uses the **<tab>** character to delineate texts from different files. **paste** writes the merged text to standard output; thus, **paste** can be used at the head of a shell pipeline.

If **paste** reads EOF from any of the input files while other files still contain data, it substitutes blank lines as input from the file that has ended.

Options

paste recognizes the following command-line options:

-d *list* Use the characters in *list* to separate the output fields. The characters in *list* are taken in sequence and used circularly, i.e., taken in order until the end of *list* is reached, then returning to the first character in *list*. By default, **paste** uses the **<tab>** character to delineate the output fields. The following character sequences have special meaning when encountered in *list*:

**** Output a single backslash character
\t Output a **<tab>** character
\n Output a **<newline>** character
\0 Output a null string (i.e., no separator between output fields)

-s Output successive lines from each input *file* across the page, with each input line separated from the next by a **<tab>** character. After all input lines from a given file have been concatenated, terminate the output line with a **<newline>** character and repeat the process on the next input file.

Example

The following two files will be used for subsequent examples. **File1** contains:

```
File1_Line1
File1_Line2
File1_Line3
File1_Line4
```

File2 contains:

```
File2_Line1
File2_Line2
File2_Line3
File2_Line4
```

The command

```
paste File1 File2
```

generates the following output:

```
File1_Line1        File2_Line1
File1_Line2        File2_Line2
File1_Line3        File2_Line3
File1_Line4        File2_Line4
```

Adding the option **-s** yields the output:

```
File1_Line1        File1_Line2        File1_Line3        File1_Line4
File2_Line1        File2_Line2        File2_Line3        File2_Line4
```

See Also

awk, commands, cut, sed

Notes

paste is copyright © 1989 by The Regents of the University of California. All rights reserved.

patch — Command

Patch a variable or flag within the kernel
/conf/patch [-k] *image symbol=value ...*

The command **patch** alters the value of datum *symbol* to *value* in executable *image*. In general, you should use **patch** to alter configuration data (constants) in programs, in device drivers, and in the COHERENT kernel. For **patch** to work with a symbolic constant, *image* must have a symbol table that includes information about *symbol*. Therefore, executables that have been processed by the command **strip** cannot be **patch**ed.

Options

patch recognizes the following command-line options:

-k Patch *image*, and patch the kernel memory of the running COHERENT system via device **/dev/kmem**. Only the superuser **root** can use **patch** to access kernel memory.

-K Patch **/dev/kmem** only. Refer to *image* for its symbol table, but do not change it.

-p "Peek" — just display current values; change nothing.

-v Verbose — display values before and after patching.

Variable Names

symbol and *value* can be either a numeric constant or a symbol from the symbol table of *image*. *symbol* and *value* expressions can include a numeric offset. In addition, *value* can be composed of the construct **makedev(***major***,***minor***)**, where *major* and *minor* are the "major" and "minor" device numbers, respectively, resulting in a **dev_t**-sized device type. No spaces can appear around the equal sign in the **symbol=***constant* construct.

Numeric constants default to decimal, but may be specifyed with a leading **0** prefix to specify an octal number or a **0x** prefix to specify a hexadecimal number.

The size of the altered *symbol* field is, by default, **sizeof(int)**. **patch** recognizes the following explicit size overrides:

:c The size of the altered field is **sizeof(char)**.

:i The size of the altered field is **sizeof(int)**.

:l The size of the altered field is **sizeof(long)**.

:s The size of the altered field is **sizeof(short)**.

Example

The following example gives technique that allows kernel display — that is, the output of the routines **cmn_err()** and the kernel's version of **printf()** — to go to a serial port. With this, you can save the panic messages and register dumps on a terminal screen or printer page while you reboot and try to track down what went wrong. To do so, plug a terminal into a serial port, and then do the following.

1. Find the major and minor numbers of a working serial port. Do not configure the port for modem control or flow control; use something simple like **com2l**. Make sure you can send data out the port; for example see that the command

   ```
   date > /dev/com2l
   ```

   sends data to the terminal's screen. The baud rate for the port will be whatever is specified for the default in file **/etc/default/async** — 9600 unless you have changed it.

2. Make sure the port is *not* enabled.

3. Create a test kernel around that you can modify. Call it something easily remembered, such as **/testcoh**.

4. Patch the kernel with the command

   ```
   /conf/patch -v /testcoh condev=makedev(major,minor):s
   ```

   where *major* is the major number for the serial port, and *minor* is its minor number.

5. Boot the patched kernel.

With this change, you will not be able to control kernel output with XON and XOFF, nor will you seen kernel output from very early startup (before the page tables are working) appear on the serial device.

Example

The following example patches the kernel to redirect error messages to a terminal device on a serial port, instead of displaying them on the console:

```
/conf/patch -v /Ikernel_name "condev=makedev(maj, min):s"
```

where *kernel_name* names the kernel you wish to patch, and *maj* and *min* are, respectively, the major and minor device numbers of the serial port to which you wish to redirect messages.

Note that **condev** is a short integer, so the ":s" is essential. The patch is made to the file on disk. You must reboot before it can work — chaos results if you try to switch console devices in a running kernel.

See Also

commands, device drivers, kernel

Notes

It is extremely dangerous to patch the COHERENT kernel. Almost all changes that you may wish to make the kernel can be accomplished more safely by using the commands **idtune** and **idmkcoh**. For details on how to use the commands, see their entries in the Lexicon. Therefore, do not use **/conf/patch** to patch the kernel unless you know *exactly* what you are doing. *Caveat utilitor!*

LEXICON

Beginning with release 4.2 of COHERENT, the symbol table has been removed from the kernel, and is kept in its own file. The symbol-table file is named after its corresponding kernel; for example, the symbol table for a kernel named **/coherent** is kept in file **/coherent.sym**. This complicates using **patch** to hot-patch a kernel. As noted above, you are well advised to use commands **idtune** and **idenable** to modify your kernel configuration, than using **patch** to hot-patch an existing kernel.

PATH — Environmental Variable

Directories that hold executable files

PATH names a default set of directories that are searched by COHERENT when it seeks an executable file. You can set **PATH** with the command **PATH**. For example, typing

```
PATH=/bin:/usr/bin
```

tells COHERENT to search for executable files first in **/bin**, and then in **/usr/bin**. Note the use of the colon ':' to separate directory names.

See Also

environmental variables, path.h

path() — General Function (libc)

Path name for a file
#include <path.h>
#include <stdio.h>
char ***path(***path***,** *filename***,** *mode***);**
char **path***,** **filename***;**
int *mode***;**

The function **path()** builds a path name for a file.

path points to the list of directories to be searched for the file. You can use the function **getenv()** to obtain the current definition of the environmental variable **PATH**, or use the default setting of **PATH** found in the header file **path.h**, or, you can define *path* by hand.

filename is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

X_OK	Execute the file
W_OK	Write to the file
R_OK	Read the file

path() calls the function **access()** to check the access status of *filename*. If **path()** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
      char *env, *pathname;
      int mode;

      if (argc != 3)
            fatal("Usage: findpath filename mode");

      if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
            fatal("modes: 1=execute, 2=write, 4=read");

      env = getenv("PATH");
      if ((pathname = path(env, argv[1], mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);
            return;
      } else
            fatal("search failed");
}
```

See Also

access(), libc, PATH, path.h

path.h — Header File

Define/declare constants and functions used with PATH
#include <path.h>

path.h declares constants used to handle the environmental variable **PATH**. These include, among others, the default path, the path separator, and the list separator. **path.h** also declares the function **path()**.

See Also

header files, path(), PATH

pathalias — Command

Generate a set of paths among computers
/usr/lib/mail/pathalias [-ivcDf] [-d *link***] [-l** *host***] [-t** *link***] [** *datafile ...* **]**

The command **pathalias** computes the shortest path and corresponding route from a host to every other known, reachable host. It reads host-to-host connectivity information from the standard input or *datafile*, then writes a list of host-route pairs onto the standard output. This command normally is used only by administrators of busy systems, to maintain the path information used by **smail**.

pathalias recognizes the following command-line options:

-c Print costs: print the path cost before each host-route pair.

-D Terminal domains: see **domains** section, below.

-d *arg* *arg* is a dead link, host, or network. If *arg* is of the form

*host-1***!***host-2*

pathalias treats the link from *host-1* to *host-2* as an extremely high-cost (i.e., dead) link. If *arg* is a single host name, **pathalias** treats that host as dead and uses it on any path only as the relay host of last resort. If *arg* names a network, the network requires a gateway.

-f First-hop cost: the printed cost is the cost to the first relay in a path, instead of the cost of the entire path. This option implies (and overrides) option **-c**.

-i Ignore case: map all host names to lower case. By default, case is significant.

-l *host* Set the name of the local host to *host*. By default, **pathalias** reads file **/etc/uucpname** to discover the name of your system.

-t *arg* Output trace information for *arg* onto the standard error.

-v Verbose: report some statistics on the standard error output.

Input Format

A line that begins with white space continues the preceding line. **pathalias** ignores anything following a '#'.

A list of a host-to-host connection consists of a **from** host in column one, followed by white space, followed by a comma-separated list of **to** hosts, called *links*. A link may be preceded or followed by a network character to use in the route. Valid network characters are '!' (default), '@', ':', and '%'. A link (and network character, if present) can be followed by a "cost" enclosed between parentheses.

The *cost* is an arithmetic expression that includes numbers, parentheses, and the operators '+', '-', '*', and '/'. It cannot be negative. **pathalias** recognizes the following symbolic costs:

LOCAL A local-area-network connection. Set cost to 25.

DEDICATED A high-speed dedicated link. Set cost to 95.

DIRECT A toll-free telephone call. Set cost to 200.

DEMAND A long-distance telephone call. Set cost to 300.

HOURLY An hourly poll. Set cost to 500.

EVENING A time-restricted telephone call. Set cost to 1,800.

DAILY A daily poll (also called **POLLED**). Set cost to 5,000.

WEEKLY An irregular poll. Set cost to 30,000.

In addition, the symbolic cost **DEAD** is a very large number (effectively, infinite); **HIGH** and **LOW** are -5 and +5, respectively, for baud-rate or quality bonuses/penalties; and **FAST** is -80, for adjusting costs of links that use high-speed modems (9600 baud or faster). These symbolic costs represent an imperfect measure of bandwidth, monetary cost, and frequency of connections. For most mail traffic, it is important to minimize the number of hosts in a route; for this reason, **HOURLY** times 24 is much larger than **DAILY**. If no cost is given, **pathalias** uses a default cost of 4,000.

For the most part, an arithmetic expression that mixes symbolic constants other than **HIGH**, **LOW**, and **FAST** makes no sense. For example, if a host calls a local neighbor whenever there is work, and in addition polls every evening, the cost is **DIRECT**, not **DIRECT+EVENING**.

Examples

Consider the following input:

```
down      princeton!(DEDICATED), tilt,
          %thrash(LOCAL)
princeton topaz!(DEMAND+LOW)
topaz     @rutgers(LOCAL+1)
```

If a link is encountered more than once, the least-cost occurrence dictates the cost and network character. **pathalias** treats links as bidirectional but asymmetric: for each link declared in the input, **pathalias** assumes a **DEAD** reverse link.

If the "to" host in a link is enclosed by angle brackets, **pathalias** regards the link as being terminal, and heavily penalizes all links beyond it. For example, when given the input

```
seismo    <research>(10), research(100), ihnp4(10)
research  allegra(10)
ihnp4     allegra(50)
```

pathalias generates a direct path from site **seismo** to site **research**; however, the path from **seismo** to **allegra** uses **ihnp4** as a relay, not **research**.

The set of names by which a host is known to its neighbors is called its *aliases*. Aliases are declared as follows:

name **=** *alias*, *alias ...*

name is the name by which the host is known to its predecessor in the route.

Fully connected networks, such as the Internet or a local-area network, are declared as follows:

net **=** {*host*, *host*, **...**}

The list of hosts may be preceded or followed by a routing character (by default, '!'), and may be followed by a cost (default 4,000). The network name is optional; if not given, **pathalias** makes one up. Consider the following input:

```
etherhosts = {rahway, milan, joliet}!(LOCAL)
ringhosts = @{gimli, alida, almo}(DEDICATED)
= {etherhosts, ringhosts}(0)
```

The routing character used in a route to a network member is the one encountered when "entering" the network. For details, see the sections on gateways and domains, below.

If you wish to give connection data, but also wish to hide the host names, use a declaration of the form:

private {*host*, *host*, **...**}

pathalias will not generate a route *to* a private host, but it may produce routes *through* it. The scope of a **private** declaration extends from the declaration either to the end of the input file in which it appears, or to a **private** declaration with an empty host list, whichever comes first. The latter scope rule lets you retain the semantics of a **private** declarations when you pass data to **pathalias** via the standard input.

Dead hosts, links, or networks may be presented in the input stream by declaring

dead {*arg*, **...**}

where *arg* has the same form as the argument to the command-line option **-d**.

To force a specific cost for a link, use

delete {*host-1***!***host-2***}**

to delete all prior declarations, then re-declare the link as desired. To delete a host and all its links, use the instruction:

delete {*host***}**

Diagnostic messages name the file in which **pathalias** found the error. To change the file's name, use the instruction:

file {*filename***}**

You can fine-tune an entry by adjusting the weights of all links from a given host. For example:

adjust {*host-1*, *host-2***(LOW),** *host-3***(-1)}**

If no cost is given, **pathalias** uses a default of 4,000.

The following script pipes into **pathalias** input from compressed (and uncompressed) files:

```
for i in $*; do
      case $i in
            *.Z)  echo "file {‘expr $i : '.Z'‘}
                  zcat $i ;;
            *)    echo "file {$i}"
                  cat $i ;;
      esac
      echo "private {}"
done
```

Output Format

pathalias writes to the standard output a list of host-route pairs, where the route is a string appropriate for use with **printf()**, e.g.:

```
rutgers  princeton!topaz!%s@rutgers
```

%s in the route string is replaced by the name of the user to whom the message is being sent. This task normally is performed by a mailer, e.g., **mail** or **elm**.

Except for domains, the name of a network is never used in routes. Thus, in the earlier example, the path from down to up would be **up!%s**, not **princeton-ethernet!up!%s**.

Gateways

pathalias represents a network by a pseudo-host and a set of network members. Links from the members to the network have the weight given in the input, whereas the cost from the network to its members is zero. If a network is declared dead, the member-to-network links are marked dead, which effectively prohibits access to the network from its members.

If, however, the input also shows an explicit link from any host to the network, then that host can be used as a gateway. In particular, the gateway need not be a network member. For example, if CSNET is declared dead and the input contains

```
CSNET = {...}
csnet-relay CSNET
```

then routes to CSNET hosts will use **csnet-relay** as a gateway.

Domains

A network whose name begins with '.' is called a *domain*. Domains are assumed to require gateways, i.e., they are **DEAD**. The route given by a path through a domain is similar to that for a network, but here the domain name is tacked onto the end of the next host. Subdomains are permitted. For example, the definition

```
harvard  .EDU   # harvard is gateway to .EDU domain
.EDU   = {.BERKELEY, .UMICH}
.BERKELEY = {ernie}
```

yields:

```
ernie   ...!harvard!ernie.BERKELEY.EDU!%s
```

Output is given for the nearest gateway to a domain. For example, the example above yields:

```
.EDU   ...!harvard!%s
```

Output is given for a subdomain if it has a different route than its parent domain, or if all its ancestor domains are private.

If the you use its command-line option **-D**, **pathalias** treats a link from a domain to a host member of that domain as terminal. This property extends to host members of subdomains, etc., and discourages routes that use any domain member as a relay.

Files

/usr/local/lib/palias.dir — Default output
/usr/local/lib/palias.pag — Default output
comp.mail.maps — Likely location of some input files

See Also

commands, mail [overview], pathmerge, smail
Honeyman P., Bellovin, S.M.: PATHALIAS, or the care and feeding of relative addresses. Atlanta, *Proceedings of the Summer USENIX Conference*, 1986.

Notes

This command is not used by the implementation of **smail** that COHERENT uses. It is included, however, for compatibility with other implementations.

The order of arguments is significant. In particular, options **-i** and **-t** should appear early.

pathconf() — System Call (libc)

Get a file variable by path name
#include <unistd.h>
long pathconf(*path***,** *name***)**
const char ******path***; int** *name***;**

pathconf() returns the value of a limit or option associated with the file *path*. *name* is a symbolic constant (defined in **<unistd.h>**) that represents the limit or option to be returned. The value that **pathconf()** returns depends upon the type of file that *path* names.

pathconf() can return information about the following constants:

_PC_LINK_MAX
The maximum value of a file's link count. If *path* names a directory, the value returned applies to the directory itself.

_PC_MAX_CANON
The number of bytes in a terminal's canonical input queue. Behavior is undefined if *path* does not name a terminal file.

_PC_MAX_INPUT
The number of bytes for which space will be available in a terminal's input queue. Behavior is undefined if *path* does not name a terminal file.

_PC_NAME_MAX
The number of bytes in a file name. The behavior is refined if *path* does not name a directory. The value returned applies to the file names within the directory.

_PC_PATH_MAX
The number of bytes in a path name. Behavior is undefined if *path* does not refer to a directory. If *path* names the current working directory, **pathconf()** returns the maximum length of a relative path name.

_PC_PIPE_BUF
The number of bytes that can be written atomically when writing to a pipe. If *path* names a pipe or FIFO, the value returned applies to the FIFO itself. If *path* names a directory, the value returned applies to any FIFOs that exist or can be created within that directory. If *path* names any other type of file, behavior is undefined.

_PC_CHOWN_RESTRICTED
chown() can be used only by a process with appropriate privileges, and only to change the group ID of a file to either that process's effective group ID or one of its supplementary group IDs. If *path* names a directory, the value returned applies to any file, other than a directory, that exists or can be created within the directory.

_PC_NO_TRUNC
Path-name components longer than **NAME_MAX** generate an error. The behavior isundefined if *path* does not refer to a directory. The value returned applies to the file names within the directory.

_PC_VDISABLE
If this value is defined, terminal-special characters can be disabled. Behavior is undefined if *path* does not name a terminal file.

The value of the system limit or option that *name* specifies does not change during the lifetime of the calling process.

pathconf() fails and returns -1 if *name* is not set to a recognized constant. It fails, returns -1, and sets **errno** to an appropriate value if any of the following is true:

- The process that calls **pathconf()** lacks permission to search a directory named in *path*. **pathconf()** sets **errno** to **EACCES**.
- *path* is needed for the command specified and it either points to an empty string or names a file that does not exist. **pathconf()** sets **errno** to **ENOENT**.
- A component of *path*'s prefix is not a directory. **pathconf()** sets **errno** to **ENOTDIR**.
- *name* is an invalid value. **pathconf()** sets **errno** to **EINVAL**.

See Also

fpathconf(), libc
POSIX Standard, §5.7.1

pathmerge — Command

Merge sorted paths files
/usr/lib/mail/pathmerge *file ...*

pathmerge reads the sorted path *file*s, each of which was generated by command **pathalias**, merges the path information they contain, and writes the result onto the standard output. It normally is used only by administrators of busy systems, to maintain the path information used by **smail**.

In its output, **pathalias** writes one path given for each host name. It gives precedence in paths to the *file*s that appear earlier in the argument list. The file name '-' represents the standard input; this lets you mingle input from files with input from the standard input.

As an example of the use of **pathmerge**, consider two files, **forces** and **paths**, whose contents, respectively, are

```
ihnp4    cbosgd!ihnp4!%s
muts12   muts12!%s
sun      sun!%s
```

and:

```
cbosgd   cbosgd!%s
ihnp4    ihnp4!%s
sun      ames!sun!%s
uunet    uunet!%s
```

The command

```
pathmerge forces paths
```

writes the following onto the standard output:

```
cbosgd   cbosgd!%s
ihnp4    cbosgd!ihnp4!%s
muts12   muts12!%s
sun      sun!%s
uunet    uunet!%s
```

For the purposes of **pathmerge**, a host name is terminated a by a space, a tab, a colon, or a newline. The number of *file*s that you can pass to **pathmerge** is limited by the number of available file descriptors, as all of the files are opened and read simultaneously.

See Also

commands, mail [overview], mkline, mkpath, mksort, mkdbm, pathalias, smail

Notes

This command is not used by the implementation of **smail** that COHERENT uses. It is included, however, for compatibility with other implementations.

Copyright © 1987, 1988 Ronald S. Karr and Landon Curt Noll. Copyright © 1992 Ronald S. Karr.

For details on the distribution rights and restrictions associated with this software, see file **COPYING**, which is included with the source code to the **smail** system; or type the command: **smail -bc**.

paths — System Administration

Routing data base for mail
/usr/lib/mail/paths

File **/usr/lib/mail/paths** holds the data base that the command **smail** uses to route mail.

Each line gives routing information to a host, and has the following format:

```
host  route [cost]
```

host names a remote host. The *route* field details the route by which mail can travel from your system to *host*. Note that it includes the **printf()**-style format string "%s". This field uses the bang-path format for describing a route. For example, if you access site **foo** via site **bar**, then route field for site **bar** reads:

```
bar   foo!bar!%s
```

smail uses the optional field *cost* to decide whether to queue mail that is spooled for other systems, or to invoke the command **uucico** to deliver the mail immediately. If the *cost* is at or below **smail**'s "queueing threshold", then **smail** attempts to deliver it immediately. This speeds mail delivery between hosts that enjoy an inexpensive UUCP link, such as a serial line; and batches mail that must be sent over expensive media, such as long-distance telephone. If the *cost* field is absent, **smail** gives this host a cost value above that of its queueing threshold.

Note that the value in the *costs* field does not override the connection times set in the UUCP file **sys**. Thus, this field is useful only for systems that you can call any time, or that you call frequently.

Example

The following gives a sample **paths** file for a COHERENT system named **lepanto**:

```
friend             friend!%s          300
hubsys             hubsys!%s          95
lepanto            %s                 0
lepanto.ampr.org   %s                 0
widget             hubsys!widget!%s   95
```

As this file shows, **lepanto** is linked to systems **hubsys** and **friend**. The cost of 95 associated with **hubsys** is low, and is appropriate to a low-cost link, such as a hard-wired link. On the other hand, the cost of 300 associated with **friend** is high, which indicates that the connection with **friend** is expensive, such as a long-distance telephone connection. If cost is 100 or greater, mail will be queued for later delivery. A cost below 100 tells **smail** to attempt immediate delivery.

In this example, machine **lepanto** is registered in the domain **ampr.org**. "ampr" is an abbreviation for "Amateur Packet Radio," which indicates that **lepanto** is a packet-radio node. Note that machine name **lepanto** appears in both conventional form ("lepanto") and domain form ("lepanto.ampr.org"); this is done to make it easier for **smail** to resolve addresses.

lepanto can use **hubsys** to forward mail to **widget**. Thus, when **smail** receives mail for system **widget**, it will transmit it to **hubsys** for forwarding. Note that **hubsys**'s administrator must have given **lepanto** permission to use it as a mail relay, or this will not work.

See Also

Administering COHERENT, mail [overview], smail

Notes

Please note that the mail-routing program **smail** does not actually read **/usr/lib/mail/paths** when it processes mail; rather, it reads a DBM-style data base that is built from this file. The DBM data base can be read much faster than an ordinary text file, thus improving the speed with which **smail** handles mail. Thus, when you edit **paths**, you must invoke the command **mkpaths** to "cook" its contents into DBM format, so **smail** see the changes you have made. For information on DBM-style data bases, see the Lexicon entry for **libgdbm**.

pattern — Definition

A **pattern** is any combination of text and wildcard characters that can be interpreted by a command. Patterns are also called "regular expressions".

The function **pnmatch()** compares two patterns and indicates whether they match.

For a fuller explanation of how to use patterns within applications, see the section on *Expert Editing* in the tutorial for the line editor **ed**.

See Also

egrep, pnmatch(), Programming COHERENT, wildcards

pause() — System Call (libc)

Wait for signal
#include <unistd.h>
int pause()

pause() suspends execution until the process receives a signal. The awaited signal could come from **kill()**, **alarm()**, or the controlling terminal.

See Also

alarm(), kill(), libc, signal(), sleep(), unistd.h
POSIX Standard, §3.4.2

pclfont — Command

Prepare a PCL font for downloading via MLP
pclfont [-f *n***]** *font* **[...** *font***]**

The command **pclfont** prepares each *font* for downloading via the MLP spooler to a printer that runs the Hewlett-Packard Page Control Language (PCL). *font* must give the full path name of a PCL bitmapped "soft font". **pclfont** brackets each *font* with the PCL commands that tell the printer to load the font into a given "slot" in its memory,

and to let the font reside permanently in memory, then writes the altered *font* to the standard output.

The option **-f** names the slot into which you want to load *font*. If the command line names more than *font*, **pclfont** sequentially assigns slots beginning with slot *n*. If you do not use the option **-f**, **pclfont** assigns slots beginning with slot 1.

The processed fonts can either be piped to the command **lp** or redirected into a file for later downloading. When downloaded via **lp**, you must use the MLP device **hpfont**. For example, to download fonts **tr240bpn.usp** and **op240bpn.usp** into slots 16 and 17 on your printer, use the command:

```
pclfont -f 16 tr240bpn.usp op240bpn.usp | lp -dhpfont
```

See Also

commands, lp, printer, troff

pclose() — STDIO Function (libc)

Close a pipe
#include <stdio.h>
int pclose(*fp***)**
FILE **fp**;

pclose() closes the pipe pointed to by *fp*, which must have been opened by the function **popen()**.

pclose() awaits the completion of the child process and performs other cleanup. It returns the value from a **WAIT** done on the child process. This value includes information in addition to the "simple" exit value of the child process.

Example

For an example of this function, see the Lexicon entry for **popen**.

Files

<stdio.h>

See Also

fclose(), fopen(), libc, pipe(), popen(), sh, system(), wait()

Diagnostics

pclose() returns -1 if *fp* had not been created by a call to **popen()**. Otherwise, **pclose()** returns the exit status of the *command*, in the format described in the entry for **wait()**: exit status in the high byte, signal information in the low byte.

perror() — General Function (libc)

System call error messages
#include <errno.h>
perror(*string***)**
char **string**; **extern int** *sys_nerr*; **extern char** **sys_errlist*[];

perror() prints an error message on the standard error device. The message consists of the argument *string*, followed by a brief description of the last system call that failed. The external variable **errno** contains the last error number. Normally, *string* is the perror of the command that failed or a file perror.

The external array **sys_errlist** gives the list of messages used by **perror()**. The external **sys_nerr** gives the number of messages in the list.

See Also

errno, errno.h, libc
ANSI Standard, §7.9.10.4
POSIX Standard, §8.1

phone — Command

Print numbers and addresses from phone directory
phone *person ...*

The command **phone** searches a number of telephone directory files for each *person* argument that is given. Any lines that matches any of the *person* arguments is printed. Typically, such lines contain the telephone number, name, and address of a person or organization. Lower-case letters in *person* can be matched by both the same letter and the corresponding upper-case letter in the phone directory.

The user may supply his own phone directory by setting the (exported) shell variable **PHONEBOOK**, to the name of that file. If given, this file is searched first. Then, the system-wide phone book is always searched.

Files

$PHONEBOOK — User-supplied phonebook (searched first)
/usr/pub/phonebook — System-wide phone directory

See Also

commands

Diagnostics

phone exits with non-zero status if a call fails. A diagnostic message is written to **stderr** if no matching entries are found.

The standard phonebook shipped with COHERENT includes telephone numbers and descriptions of third-party vendors who sell software for COHERENT. If you're looking for software to run under COHERENT, check there first.

pipe — Definition

A *pipe* directs the output stream of one program into the input stream of another program, thus coupling the programs together. With pipes, two or more programs (or *filters*) can be coupled together to perform complex transforms on streams of data. For example, in the following command

```
cat DATAFILE1 DATAFILE2 | sort | uniq -d
```

the filter **cat** opens two files and prints their contents. Its output is piped to the filter **sort**, which sorts it. The output of **sort** is piped, in turn, to the filter **uniq**, which (with the **-d** option) prints a single copy of each line that is duplicated within the file. Thus, with this simple set of commands and pipes, a user can quickly print a list of all lines that appear in both files.

See Also

filter, mkfifo(), named pipe, pipe(), Using COHERENT

pipe() — System Call (libc)

Open a pipe
#include <unistd.h>
int pipe(*fd***)**
int *fd***[2];**

A *pipe* is an interprocess communication mechanism. **pipe()** creates a pipe, typically to construct pipelines in the shell **sh**.

pipe() fills in *fd***[0]** and *fd***[1]** with *read* and *write* file descriptors, respectively. The file descriptors allow the transfer of data from one or more writers to one or more readers. Pipes are buffered to 5,120 bytes. If more than 5,120 bytes are written into the pipe, the **write()** call will not return until the reader has removed sufficient data for the **write()** to complete. If a **read()** occurs on an empty pipe, its completion awaits the writing of data.

When all writing processe close their write file descriptors, the reader receives an end of file indication. A write on a pipe with no remaining readers generates a **SIGPIPE** signal to the caller.

pipe() is generally called just before **fork()**. Once the parent and child processes are created, the unused file descriptors should be closed in each process.

Example

The following example prints the word **Waiting** until a line of data is entered. It illustrates how to use **pipe()**, **fstat()**, and **fork()**.

```
#include <stdio.h>
#include <sys/stat.h>         /* for stat */
#include <sgtty.h>            /* for stty/gtty functions */
#include <unistd.h>

static int fd[2];       /* pipe array */

main()
{
      printf("This prints 'Waiting' every second until a 'q' is hit.\n");

      /*
       * Pipe may also be constructed by /etc/mknod
       * If it is desired to have tasks communicate where
       * they are not parent and child. In this case make
       * sure the constructed pipe has the correct owner and
       * permissions. Such pipe may be used exactly like this
       * but open()ed on each side.
       */

      if (-1 == pipe(fd)) {
            fprintf(stderr, "Cannot open pipe\n");
            exit(EXIT_FAILURE);
      }

      if (fork())
            parentProcess();
      else
            childProcess();
      exit(EXIT_SUCCESS);
}

parentProcess()
{
      struct stat s;
      char buff;

      for (buff = ' '; 'q' != buff;) {
            fstat(fd[0], &s); /* get status of pipe */
            if (s.st_size) {        /* char in the pipe */
                  read(fd[0], &buff, sizeof(buff));
                  printf("Got a '%c'\n", buff);
                  continue;
            }

            /*
             * This can be any process, it can use system()
             * or exec()
             */
            printf("Waiting\n");
            sleep(1);
      }
}

childProcess()
{
      struct      sgttyb      os, ns;
      char buff;

      gtty(fileno(stdin), &os);     /* save old state */
      ns = os;                /* get base of new state */
      ns.sg_flags |= RAW;           /* process each character as entered */
      ns.sg_flags &= ~(ECHO|CRMOD); /* no echo for now... */
      stty(fileno(stdin), &ns);     /* set mode */

      do {
            buff = getchar(); /* wait for the keyboard */
            write(fd[1], &buff, sizeof(buff));
      } while ('q' != buff);
```

```
      stty(fileno(stdin), &os);     /* reset mode */
}
```

See Also

close(), libc, libsocket, mkfifo(), mknod(), read(), sh, signal(), unistd.h, write()
POSIX Standard, §6.1.1

Diagnostics

pipe() returns zero on successful calls, or -1 if it could not create the pipe.

If it is necessary to create a pipe between tasks that are not parent and child, use **/etc/mknod** to create a named pipe. These named pipes can be opened and used by different programs for communication. Remember to give them the correct owner and permissions.

If you attempt to open a pipe write only, **O_NDELAY** is set, and there are currently no readers on this pipe, **open()** returns immediately and sets **errno** to **ENXIO**.

pnmatch() — String Function (libc)

Match string pattern
int pnmatch(*string***,** *pattern***,** *flag***)**
char ******string***,** ******pattern***; int** *flag***;**

pnmatch() matches *string* with *pattern*, which is a regular expression. The shell **sh** uses patterns for file name expansion and **case** statement expressions.

pnmatch() returns one if *pattern* matches *string*, and zero if it does not. Each character in *pattern* must exactly match a character in *string*; however, the wildcards '*', '?', '[' and ']', and '[!' and ']' can be used in *pattern* to expand the range of matching.

flag must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '' and '$' can also be used in *pattern*.

Example

For an example of this function, see the entry for **fgets()**.

See Also

egrep, grep, libc, sh, string.h, wildcards

Notes

flag must be zero or one for **pnmatch()** to yield predictable results.

pnmatch() is a more powerful version of the ANSI functions **strstr()** and **strcmp()**.

For an **egrep**-style version of **pnmatch()**, see the function **regexp()**. It is described in the Lexicon article **libmisc**.

pointer — C Language

A *pointer* is an object whose value is the address of another object. The name "pointer" derives from the fact that its contents "point to" another object. A pointer may point to any type, complete or incomplete, including another pointer. It may also point to a function, or to nowhere.

The term *pointer type* refers to the object of a pointer. The object to which a pointer points is called the *referenced type*. For example, an **int** * ("pointer to **int**") is a pointer type; the referenced type is **int**. Constructing a pointer type from a referenced type is called *pointer type derivation*.

The Null Pointer

A pointer that points to nowhere is a *null pointer*. The macro **NULL**, which is defined in the header **stdio.h**, defines the null pointer. The null pointer is an integer constant with the value zero. It compares unequal to a pointer to any object or function.

Declaring a Pointer

To declare a pointer, use the indirection operator '*'. For example, the declaration

```
int *pointer;
```

declares that the variable **pointer** holds the address of an **int**-length object. Likewise, the declaration

```
int **pointer;
```

declares that **pointer** holds the address of a pointer whose contents, in turn, point to an **int**-length object.

Failure to declare a function that returns a pointer will result in that function being implicitly declared as an **int**. This does not cause an error on microprocessors in which an **int** and a pointer have the same size; however, if you transport this code to a microprocessor in which an **int** consists of 16 bits and a pointer consists of 32 bits, the pointer will be truncated truncated to 16 bits and the program probably will fail.

C allows pointers and integers to be compared or converted to each other without restriction. The COHERENT C compiler flags such conversions with the strict message

```
integer pointer pun
```

and comparisons with the strict message

```
integer pointer comparison
```

These problems should be corrected if you want your code to be portable to other computing environments.

See **C language** for more information.

Wild Pointers

Pointers are omnipresent in C. C also allows you to use a pointer to read or write the object to which the pointer points; this is called *pointer dereferencing*. Because a pointer can point to any place within memory, it is possible to write C code that generates unpredictable results, corrupts itself, or even obliterates the operating system if running in unprotected mode. A pointer that aims where it ought not is called a *wild pointer*.

When a program declares a pointer, space is set aside in memory for it. However, this space has not yet been filled with the address of an object. To fill a pointer with the address of the object you wish to access is called *initializing* it. A wild pointer, as often as not, is one that is not properly initialized.

Normally, to initialize a pointer means to fill it with a meaningful address. For example, the following initializes a pointer:

```
int number;
int *pointer;
   . . .
pointer = &number;
```

The address operator '&' specifies that you want the address of an object rather than its contents. Thus, **pointer** is filled with the address of **number**, and it can now be used to access the contents of **number**.

The initialization of a string is somewhat different than the initialization of a pointer to an integer object. For example,

```
char *string = "This is a string."
```

declares that **string** is a pointer to a **char**. It then stores the string literal **This is a string** in memory and fills **string** with the address of its first character. **string** can then be passed to functions to access the string, or you can step through the string by incrementing **string** until its contents point to the null character at the end of the string.

Another way to initialize a pointer is to fill it with a value returned by a function that returns a pointer. For example, the code

```
extern char *malloc(size_t variable);
char *example;
   . . .
example = malloc(50);
```

uses the function **malloc** to allocate 50 bytes of dynamic memory and then initializes **example** to the address that **malloc** returns.

Reading What a Pointer Points To

The indirection operator '*' can be used to read the object to which a pointer points. For example,

```
int number;
int *pointer;
   . . .
pointer = &number;
   . . .
printf("%d\n", *pointer);
```

uses **pointer** to access the contents of **number**.

When a pointer points to a structure, the elements within the structure can be read by using the structure offset operator '->'. See the entry for **operators** for more information.

Pointers to Functions

A pointer can also contain the address of a function. For example,

```
char *(*example)();
```

declares **example** to be a pointer to a function that returns a pointer to a **char**.

This declaration is quite different from:

```
char **different();
```

The latter declares that **different** is a function that returns a pointer to a pointer to a **char**.

The following demonstrates how to call a function via a pointer:

```
(*example)(arg1, arg2);
```

Here, the '*' takes the contents of the pointer, which in this case is the address of the function, and uses that address to pass to a function its list of arguments.

A pointer to a function can be passed as an argument to another function. The functions **bsearch** and **qsort** each take a function pointer as an argument. A program may also use arrays of pointers to functions.

void *

void * is the generic pointer; it replaces **char** * in that role. A pointer may be cast to **void** * and then back to its original type without any change in its value. **void** * is also aligned for any type in the execution environment. Please note that COHERENT's C compiler does not yet recognize the type **void** *.

In Kernighan and Ritchie C, character pointers are equivalent to **void** *. To convert a program to use **void** *, rewrite the sources so that instances of

```
char *foo(bar);
```

is replaced by:

```
VOID_T *foo(bar);
```

Be sure that you do not replace legitimate **char** *s — that is, pointers that actually point to character strings. Then put the code

```
#if defined(__ANSI__) || defined(__GNUC__)
typedef void VOID_T
#else
typedef char VOID_T
#endif
```

into an application-owned header file that is included by every source file.

Pointer Conversion

One type of pointer may be converted, or *cast*, to another. For example, a pointer to a **char** may be cast to a pointer to an **int**, and vice versa.

The ANSI Standard states that any pointer can be cast to type **void** * and back again without its value being affected in any way. (Once again, please note that COHERENT's C compiler does not yet recognize the type **void** *.) Likewise, any pointer of a scalar type may be cast to its corresponding **const** or **volatile** version. The qualified pointers are equivalent to their unqualified originals.

Pointers to different data types are compatible in expressions, but only if they are cast appropriately. Using them without casting produces a *pointer-type mismatch*. The translator should produce a diagnostic message when it

detects this condition.

Pointer Arithmetic

Arithmetic may be performed on all pointers to scalar types, i.e., pointers to **char**s or **int**. Pointer arithmetic is quite limited and consists of the following:

1. One pointer may be subtracted from another.
2. An **int** or a **long**, either variable or constant, may be added to a pointer or subtracted from it.
3. The operators **++** or **--** may be used to increment or decrement a pointer.

No other pointer arithmetic is permitted. No arithmetic can be performed on pointers to non-scalar objects, e.g., pointers to functions.

When an **int** or **long** is added to a pointer, it is first multiplied by the length of what the pointer is declared as pointing to. Thus, if a pointer to an **int** is incremented by two, it points down two more **int**s, not two more characters. The following program demonstrates this feature:

```
char *pc = "Welcome";
int array[5] = { 1, 2, 3, 4, 5 };
int  *pi = array;

main()
{
       pc += 2;       /* pc points to  'l' */
       pi += 2;       /* pi points to 3 */
}
```

See Also

C language data formats **operators, portability, Programming COHERENT**
ANSI Standard, §6.1.2.5, §6.2.2.1, §6.2.2.3, §6.3.2.2-3, §6.5.4.1

poll() — System Call (libc)

Query several I/O devices
#include <poll.h>
int poll(*fds***,** *nfds***,** *timeout***)**
struct pollfd *fds***[]; unsigned long** *nfds***; int** *timeout***;**

The COHERENT system call **poll()** polls one or more file streams for one or more polling conditions. *fds* gives the address of an array of **struct**s of type **pollfd**, which has the following structure:

```
struct pollfd {
      int   fd;   /* file descriptor */
      short events;     /* requested events */
      short revents;    /* returned events */
};
```

Field **fd** gives the file descriptor for a file stream, as returned by a call to **open()**, or **creat()**. Fields **events** and **revents** give, respectively, the polling conditions that interest you, and those that have occurred. The legal conditions, as defined in header file **poll.h**, are as follows:

POLLIN Input, or a non-priority or file-descriptor passing message, is available for reading. In **revents**, this bit is mutually exclusive with **POLLPRI**.

POLLPRI A priority message is available for reading. In **revents**, this bit is mutually exclusive with **POLLIN**.

POLLOUT Output may be performed; the output queue is not full.

POLLERR An error message has arrived. This field is used only in **revents**, and is ignored in **events**.

POLLHUP A hangup has occurred. This field is used only in **revents**, and is ignored in **events**.

POLLNVAL The specified **fd** value does not belong to an open I/O stream. This field is used only in **revents**, and is ignored in **events**.

nfds gives the number of entries in *fds*.

For each array element *fds***[***i***]**, **poll()** examines the file descriptor *fds***[***i***].fd** for the events specified by bits set in *fds***[***i***].events**, and places the resulting status into *fds***[***i***].revents**. If the **fd** value is less than zero, **revents** for that

entry is set to zero. Event flags **POLLIN**, **POLLPRI**, and **POLLOUT** are set in **revents** only if the same bits are set in **events** *and* the corresponding condition holds. Event flags **POLLHUP**, **POLLERR**, and **POLLNVAL** are always set in **revents** if the corresponding condition holds, regardless of the contents of **events**.

If none of the defined events for any of the file descriptors has occurred, **poll()** waits for *timeout* milliseconds. Because the system clock runs at 100 hertz, the value used for *timeout* is the next higher multiple of ten milliseconds. If *timeout* is zero, **poll()** returns immediately. If *timeout* is -1, **poll()** blocks until a requested event occurs or a signal interrupts the call.

poll() returns the number of file descriptors for which **revents** is nonzero. It returns zero if it timed out with no matching events. If the call failed, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of using **poll()** to read a serial port, see the Lexicon entry for **ioctl().** The following example uses **poll()** to sleep for a fraction of a second.

```
#include <poll.h>
#include <sys/v_types.h>
#include <sys/times.h>

main()
{
      struct pollfd fds;
      int timeout;
      struct tms tmp;
      int before; /* time in millisec before poll() */
      int after; /* time in millisec after poll() */

      timeout = 270; /* sleep time is timeout * 10 millisec */

      fds.fd = -1; /* no file needed for sleeping */

      before = times(&tmp); /* Get time before poll */

      /* sleep not less than 0.270 sec */
      poll(&fds, 1, timeout);

      after = times(&tmp); /* Get time after poll */

      printf("%d\n", (after - before) * 1000 / CLK_TCK);
}
```

See Also

libc, poll.h

poll.h — Header File

Define structures/constants used with polling devices
#include <poll.h>

poll.h defines structures and constants used by routines that poll devices.

See Also

header files

popd — Command

Pop an item from the directory stack
popd [*item ...* **]**

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **popd** pops an item from the directory stack. If called without an argument, it pops the last item. Otherwise, it pops the given stack *item*s in the order requested, where each *item* is a positive integer and zero is the top of the stack.

See Also

commands, dirs, pushd, sh

popen() — STDIO Function (libc)

Open a pipe
#include <stdio.h>
FILE *popen(*command, how***)**
char ****command, *how*;**

popen() opens a pipe. It resembles the function **fopen()**, except that the opened object is a command line to the shell **sh** rather than a file.

The caller can read the standard output of *command* when *how* is **r**, or write to the standard input of *command* when *how* is **w**. **popen()** returns a pointer to a **FILE** structure that may be read or written.

Example

This example is equivalent to the command

```
ls -l | mail me
```

where **me** is your login identifier.

```
#include <stdio.h>
main()
{
      FILE *ifp, *ofp;
      int c;

      if ((NULL == (ofp = popen("lmail me", "w"))) ||
          (NULL == (ifp = popen("ls -l",     "r")))) {
                fprintf(stderr, "cannot popen\n");
            exit(1);
      }

      while (EOF != (c = fgetc(ifp)))
            fputc(c, ofp);

      pclose(ifp);
      pclose(ofp);
}
```

Files

<stdio.h>

See Also

fclose(), fopen(), libc, pclose(), pipe(), sh, system(), wait()

Diagnostics

popen() returns NULL if the link to *command* could not be established.

port — System Administration

File that describes ports for UUCP
/usr/lib/uucp/port

File **/usr/lib/uucp/port** names and describes the serial ports that **uucico** and **cu** use to connect to remote systems.

port consists of a set of entries, one for each port. Entries should be separated from each other by one blank line. Each entry consists of one or more of the following commands:

port *port_name*
Name the port being described. This command must appear first in every port's entry.

type *string*
This command gives the type of port. It must appear immediately after the **port** command. *string* must be one of the following:

direct The port directly accesses another, usually via a serial port.

modem
The port accesses a modem. This is the default.

pipe The connection is a pipe that runs through another program

stdin The connection runs through the standard input and standard output. Use this option when **uucico** is run as a login shell

tcp The port is a TCP port.

protocol *string*
List the protocols that can be used with this port. If **/usr/lib/uucp/sys** contains a list of protocols, that list takes precedence over the one set in **port**. We recommend that protocols be specified in the file **sys** intsead of here. For information on the available protocols, see the Lexicon article **sys**.

protocol-parameter *protocol parameter*
Set a *parameter* for the *protocol*. This command recognizes exactly the same arguments as its namesake in the system-configuration file **sys**. For information on how to use this command, see the Lexicon entry for

seven-bit true|false
If **true**, then this port (or the modem plugged into it) supports only seven-bit transfers; if false, then it supports both seven-bit and eight-bit protocols. **uucico** uses this command only during protocol negotiation, to force the selection of a protocol that works across a seven-bit link. It will not prevent eight-bit characters from being transmitted. The default is **false**.

Note that some devices use only seven bits to define a character, and reserve the eighth bit as a parity bit. It is not possible it is not possible to send eight-bit characters across such devices.

reliable true|false
This command is used only when your system negotiates with the remote system over what protocol to use. If set to **false**, it forces your system to accept only a protocol that works over a seven-bit (or unreliable) connection. If **true**, then an eight-bit protocol is acceptable. The default is false.

half-duplex true|false
If **true**, then this port supports only half-duplex communications, which forces **uucico** not to use a bidirectional protocol with this port. If it is **false**, then the port supports both half-duplex and full-duplex communications. The default is **false**. **sys**.

device *string*
This command names the device associated with the port. For example, the command

```
device /dev/com2l
```

names port **com2l** as the device used by this port. This command is used only with ports of types **modem** or **direct**.

baud *number*
speed *number*
Set the baud rate for this port. If an entry in file **/usr/lib/uucp/sys** specifies a speed but no port entry, **uucico** tries every entry in **port** that has a matching baud rate, in the order in which they appear, until it finds one that is unlocked. These commands are used only with ports of type **modem** or **direct**.

baud-range *low high*
speed-range *low high*
Set the range of speeds at which this port can be run. *low* gives the minimum speed, *high* the maximum. This command applies only to ports of type **modem**.

carrier true|false
If **true**, the port supports carrier; if **false**, the port does not. If a port does not support carrier, the carrier-detect signal will never be required on this port, regardless of what the modem chat script says. If a direct port supports carrier, the port will be set always to expect carrier.

This command applies only to ports of type **direct** or **modem**. The default for a **modem** port is true; but for a **direct** port is false.

hardflow true|false
If **true**, turn on hardware flow control for this port; otherwise, do not. The default is **true**. This command applies only to ports of type **direct** or **modem**.

LEXICON

dial-device *device*
Send dialing instructions to *device*, instead of the the normal port device. This applies only to ports of type **modem**.

dialer *string*
Names the dialer to use. Information about the dialer is read from file **/usr/lib/uucp/dial**. This applies only to ports of type **modem**.

dialer *string ...*
Execute a simple dialing script. This command can be used in situations where the dialing script is so simple that it would be cumbersome to embed it within a separate file. If the command **dialer** is used with only one argument (to name a dialing script), this command is ignored. This applies only to ports of type **modem**.

dialer-sequence *dialer phone_number ...*
Name pairs of dialers and telephone numbers. The telephone number is substituted for the escape sequences **\D** or **\T** in the *dialer* entry. In effect, this lets you name a sequence of chat scripts to use. At present, this command is the only way to use a chat script with a TCP port.

This command applies only to ports of type **modem** or **tcp**.

lockname *name*
Use *name* when locking this port. This applies only to ports of type **modem** or **direct**.

service *service_name*
Name the TCP port to use. If this names a service, then **uucico** looks the port for that service in file **/etc/services**. If it is a number, then **uucico** binds itself to that TCP port. If this command is not used, then **uucico** by default uses the well-known port 540. This command applies only to ports of type **tcp**.

command *command* **[** *arguments* **]**
If the port is of type **pipe**, name the command and its arguments with which **uucico** will be exchanging data. For example, if your system is on a network, then *command* could a form of the command **rlogin**, which would permit **uucico** to log into the remote system via the network.

Example

The following gives a sample entry for a port:

```
port MWCBBS
type modem
device /dev/com2l
baud 9600
dialer tbfast
```

The following describes each command in detail:

port This names the port being described in this entry, in this case **MWCBBS**.

type The type of port — in this case, a modem.

device The device used by this port. The device name usually matches the port name, but it does not have to.

baud The speed of the port, in this case 9600.

dialer The type of dialing device (i.e., modem) plugged into this port — in this case, the dialer named **tbfast**. This dialer is described in the file **/usr/lib/uucp/dial**. For information on how a dialer is described in its file, see the Lexicon entry for **dial**.

See Also

Administering COHERENT, dial, sys, UUCP

Notes

Only the superuser **root** can edit **/usr/lib/uucp/port**.

The file **port** supports many commands in addition to the ones described here. This article describes only those commands that might be used in typical UUCP connections. For more information, see the original Taylor UUCP documentation, which is in the archive **/usr/src/alien/uudoc.tar.Z**.

portability — Definition

Portability means that code can be recompiled and run under different computing environments without modification. Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

- Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**. If a function returns a pointer, declare it so.
- Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types.
- Do not write routines that play tricks with a machine's "magic characters"; for example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.
- Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
- Use header files to hold all machine-dependent declarations and definitions.
- Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value; this avoids unforeseen problems with undefined return values.
- Do not assume that integers and pointers have the same size or even the same kind of structure. Do not assume that pointers are all the same or can point anywhere. On the i8086, in SMALL model a pointer to a function addresses relative to the code segment, whereas a pointer to data addresses relative to the data segment. On some machines, character pointers are of a different size or structure than word pointers.
- The constant NULL is defined as being different from any valid pointer. Use it and nothing else for that purpose.
- Keep test scripts, preferably at the function level. That is, follow each function with an

  ```
  #ifdef TEST
  ```

  section that will exercise that function. Running these can rapidly isolate portability problems.
- Place plenty of

  ```
  #assert
  ```

  statements in your programs. These can often pick up portability problems.

See Also

header files, pointer, Programming COHERENT, void

POSIX Standard — Definition

The term "POSIX Standard" refers to the Portable Operating System Interface (POSIX) standard, published by the International Standards Organization (ISO) in 1990 as its standard 9945-1. It is based on standard 1003.1 published in 1988 by the Institute for Electrical and Electronics Engineers (IEEE).

The POSIX Standard is built upon the documentation for the UNIX Operating System published originally by AT&T Bell Laboratories (now by Unix Systems Laboratories). It defines a common set of guidelines to which UNIX and UNIX-like operating systems like COHERENT should adhere in order to ensure common functionality, and to maximize the portability of code from one operating system to another. The publication of the POSIX Standard is a long step towards maintaining the openness of the UNIX family of operating systems.

ANSI Standard, Programming COHERENT

pow() — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
#include <mprec.h>
void pow(*a, b, m, c***)**
mint **a, *b, *m, *c;*

pow() sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to the power of the value pointed to by *b*, reduced modulo of the value pointed to by *m*.

See Also

libmp

pow() — Mathematics Function (libm)

Compute a power of a number
#include <math.h>
double pow(*z*, *x***)**
double *z*, *x*;

pow() returns *z* raised to the power of *x*, or *z*^x. If an overflow error occurs (that is, you attempt to compute a number that is too large to fit into a double-precision floating-point number), **pow()** returns a huge value and sets **errno** to **ERANGE**.

Example

For an example of this function, see the entry for **log10()**.

See Also

libm
ANSI Standard, §4.5.5.1
POSIX Standard, §8.1

pr — Command

Paginate and print files
pr [*options* **] [** *file …***]**

pr paginates each *file* and writes it onto the standard output. At the top of each page, **pr** writes a header that that gives today's date, the file's name, the number of the page, and the number of the line in the input file at which printing begins.

The file name '-' tells **pr** to read the standard input; this lets you mingle text from one or more files with text you type from the keyboard or pipe in from another program. **pr** also reads the standard input by default if its command line does not name a file.

pr recognizes the following command-line options:

+ *skip* Skip the first *skip* pages of each input file.

-*N* Print the text in *N* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed in quotation marks.

-l*N* Set the page length to *N* lines (default, 66).

-m Print the texts simultaneously, in separate columns. Each text will be assigned an equal amount of width on the page, and any lines longer than that width will be truncated. You can use this to print several similar texts or listings simultaneously.

-n Number each line.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left-justified in a fixed-width field.

-t Suppress the printing of the header on each page, and the header and footer space.

-w*N* Set the page width to *N* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 254 columns.

See Also

cat, commands, nroff, prps

Notes

pr generates normal ASCII text, suitable for displaying on your screen or printing with a dot-matrix printer. The command **prps** also paginates text, but its output is in the PostScript language, suitable for printing on a PostScript printer.

prep — Command

Produce a word list
prep [-dfp] [-i *ifile* **] [-o** *ofile* **] [** *file ...* **]**

The command **prep** prepares a word list that is useful for statistical processing from the textual data found in each input *file*. If no *file* is given, **prep** reads the standard input for text.

For the purposes of **prep**, a word consists of a string of alphabetic letters and apostrophes. Words are written, one per line, to the standard output. Hyphenated words are treated as two words. However, any word hyphenated between two lines is rejoined as one word.

prep recognizes the following options:

-d Print a sequence number (of words in the input text) before each output word.

-f Fold upper-case letters into lower case. This is sometimes useful for producing unique lists of words.

-i *ifile* Ignore words found in *ifile*. *ifile* has words one per line that are matched against each input word, independent of case.

-o *ofile* Print only words found in *ofile*. Only one of **-i** or **-o** may be specified.

-p In addition to printing words, also print each punctuation character (printable, non-numeric characters that separate words), one per line. These lines are not counted for **-d.**

See Also

commands, deroff, ksh, sh, sort, spell, typo, wc

Notes

What constitutes a *word* is different in **deroff**, **prep**, and **wc**.

print — Command

Echo text onto the standard output
print [-enru*n***] [***argument ...***]**

The command **print** is built into the Korn shell **ksh**. It echoes each *argument* onto the standard output. Arguments are separated from each other by whitespace, and the list of arguments is terminated by a newline character.

print recognizes and substitutes for the following C-style escape sequences:

\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\0*nnn*	*nnn* is the octal value of the desired character

print recognizes the following options:

-e Re-enable expansion of C escape sequences.

-n Suppress printing of a newline at the end of the list of arguments.

-r Suppress expansion of C escape sequences.

-u*n* Redirect output from the standard output to shell file descriptor *n*.

See Also

commands, echo, ksh

printer — Technical Information

How to attach and run a printer

A *printer* is the device that transfers text to paper. The COHERENT system includes a system for spooling a file to one or more printers. *Spooling* means that the file is copied into a special area and printed by a daemon. With a spooler, more than one user can send files to the same printer at the same time, yet the files will not collide.

COHERENT also includes commands to prepare text for printing a variety of printers. These include line printers (that is, dot-matrix printers), Epson-compatible printers, laser printers that use the PCL page-description language, and printers that use PostScript. With COHERENT, you can run prepare text into a variety of formats, and print the output on any number of printers plugged into either parallel or serial ports.

COHERENT has implemented spooling in two ways. Versions of COHERENT prior to release 4.2 control printing through a version of the Berkeley command **lpr**. COHERENT release 4.2 and subsequent releases also control printing through the MLP print spooler, which implements a version of the System-V command **lp** and related tools. These systems differ greatly; each set is discussed in its own section below.

Before we begin to describe printing, please note that one major source of confusion for users is the fact that the same names occur over and over again. For example, please do not confuse the parallel-port's device driver **lp** with the print-spooler command **lp** or with the device **/dev/lp**. COHERENT inherits much of this confusion from the UNIX operating system; but we will do our best to make these terms clear to you. *Caveat lector.*

Device Drivers

Both the **lpr** and **lp** spoolers work through COHERENT's device drivers for the serial and parallel ports. The following gives an overview of these drivers.

The driver **lp** manages parallel ports. The architecture of the PC permits your computer to have up to three parallel ports. Devices **/dev/lpt1**, **/dev/lpt2**, and **/dev/lpt3** control, respectively, parallel ports 1, 2, and 3 in cooked mode. For more information, see the Lexicon entry for the driver **lp**.

COHERENT uses the driver **asy** to manage all serial ports, whether COM ports or multi-port cards. For details, see its entry in the Lexicon.

Finding the Port

Both spooler systems require that you be able to identify a port when you plug a printer into it. This can be more difficult than it seems, largely because the labels on your system's port may not be reliable: those labels reflect what MS-DOS thinks the ports are, and that may not be accurate.

The following describes how to identify the port into which you have just plugged a printer. Note that these directions assume that you are printing to a parallel port; however, you can adapt them to serial ports as well, depending on the configuration of serial devices on your system.

1. Plug the printer into an unused port. Load paper into the printer and turn it on.
2. Log in as the superuser **root**.
3. **cd** to directory **/dev**.
4. Send some output to each parallel port. The output must be something that your printer can print. If your new printer is a line printer, type:

   ```
   cat /etc/uucpname | pr > lpt1
   ```

   If the printer is a laser printer that uses PCL, type:

   ```
   cat /etc/uucpname | hp > lpt1
   ```

   Or, if the printer is a PostScript printer, type:

   ```
   cat /etc/uucpname | prps > lpt1
   ```

   If text appears on your printer, then you have discovered the correct port. Jot down its name on a piece of paper, e.g., "lpt1". If nothing happens, try the command again for **lpt2** and **lpt3**, until you have found the correct port and noted its name.
5. Exit from superuser status.

The lpr Printing System

Versions of COHERENT prior to release 4.2 use a version of the Berkeley command **lpr** to control printing. Although this command can print text onto printers plugged into either serial or parallel ports, they are almost always used through parallel ports; therefore, the descriptions in this section assume that all printers are plugged into parallel ports.

To begin, **lpr** is actually a family of commands, as follows:

hpd	Daemon that prints files on the laser printer
hpr	Spool a file for printing on the laser printer
hpskip	Abort/restart printing a file on the laser printer
lpd	Daemon that prints files on the line printer
lpr	Spool a file for printing on the line printer
lpskip	Abort/restart printing a file on the line printer

Each command has its own entry in the Lexicon, which describes it in detail.

The commands **lpr** and **hpr** dispatch text to printers: **lpr** to the printer plugged into device **/dev/lp**, and **hpr** to the printer plugged into device **/dev/hp**. Each of these devices is actually a link to the correct parallel port — that is, to devices **/dev/lpt1**, **/dev/lpt2**, or **/dev/lpt3**, as described above. (For information on what a *link* is, see the Lexicon entry for the command **ln**). The fact that each command uses a "generic" device for its output makes it easy for you to dispatch files to the right device; however, it also means that you can have only one line printer and one laser printer plugged into your computer.

When you installed COHERENT, the installation program tried to link **/dev/lp** and **/dev/hp** for you automatically; however, you may need to set them yourself (say, because you have purchased a new printer).

To set these links correctly, first follow the directions given above to identify the port into which you have plugged the printer. Then, link that port to the device by which you will access the printer. If you are installing a line printer that you will access via the command **lpr**, then you must use the command **ln** to link the port to device **/dev/lp**; if, however, the printer is a laser printer that you will access via the command **hpr**, then you must link the port to device **/dev/hp**. For example, if you have plugged a line printer into port **lpt1**, then use the following commands:

```
ln -f lpt1 lp
ln -f rlpt1 rlp
```

(Please note that the last character in "lpt1" and "rlpt1" is the numeral one — not a lower-case el.) If, however, you have plugged a laser printer into port **lpt3**, then use the following commands:

```
ln -f lpt3 hp
ln -f rlpt3 rhp
```

After you have made the links, use the command **lpr** or **hpr** (whichever is applicable) to test whether you have set up the links correctly. If you have not, go through the above procedure again.

The following describes how to use the **lpr** family of commands to print to a variety of printers.

Dumb Printers

To print on a line printer, simply use the command **lpr**. This command performs some formatting on the file, and invokes the line-printer daemon **lpd** to spool the file for printing. For example, to print the name of your system, use the command:

```
cat /etc/uucpname | pr | lpr -B
```

The option **-B** suppresses the printing of a banner page.

You can also print the output of the text-formatting command **nroff** on a line printer, assuming that your line printer understands how to backspace. For example, the manual pages included with COHERENT were formatted with **nroff**. To print the text of this Lexicon entry on your line printer, type:

```
man printer | lpr -B
```

Epson-Compatible Printers

The command **epson** massages text into a form that uses some of the text-formatting features of the Epson MX-80 printer and clones thereof. It is especially to be used with text that has been formatted with **nroff**: it turns **nroff**'s character-backspace-character sequence into the Epson escape sequences for emphasized text and italics. **epson** writes its formatted output to the standard output, from which you can pipe it to a spooler or other program.

For example, to print this manual page on an Epson-compatible printer, type:

```
man printer | epson | lpr -B
```

Laser Printers with PCL

The Hewlett-Packard LaserJet, and its clones, use the Hewlett-Packard Control Language (HPCL) to control their behavior. Note that some laser printers, such as the Apple LaserWriter, use PostScript instead of HPCL; these printers are described below.

The command **hp** prepares files to be printed on a HPCL printer. (Please do not confuse this with the device **/dev/hp**.) You should use it to prepare simple text, such as program listings, for printing on your laser printer.

Like the command **epson**, **hp** massages the output of **nroff** into escape sequences used by a printer — in this case, escape sequences used by a printer that's running the Hewlett-Packard Page Control Language (PCL). For example, to print this manual page on your PCL printer, type:

```
man printer | hp | hpr -B
```

The command **hpr** spools files to be printed on a laser printer. It works like the command **lpr**, except that it includes a number of special features; for example, you can use it to download LaserJet "soft fonts" into your printer.

PostScript Printers

Some laser printers use PostScript instead of HPCL to control their behavior. These printers expect their input to a program written in the PostScript language; if you send them ordinary text, they simply hang. To print ordinary text on a PostScript printer use the command **prps**, which is a PostScript version of the COHERENT command **pr**. It paginates text, draws a box around the page, and prints a simple header at the top of each page. For example, to print this manual page on a PostScript printer, use the command:

```
man printer | prps | hpr -B
```

Note that to print on a PostScript printer, you must use the **-B** option to the command **hpr**. If you do not, **hpr** will attempt to print a banner page in ordinary text on your printer, and your printer will hang.

The lp Printing System

Versions of COHERENT beginnning with release 4.2 also include the MLP spooler, which is an implementation of the System-V **lp** family of printing commands (hereafter called **lp**).

lp is considerably more sophisticated than the **lpr** commands. It permits you to have multiple printers of the same type (instead of just one laser printer and one line printer, as under **lpr**), which can be plugged into serial or parallel ports. It supports prioritization of printing jobs (that is, you can give some users or some types of jobs higher priority than others), lets each user set a default printer for his jobs, allows users to reprint their jobs easily, and allows applications to customize their output to take advantage of special printer features. It even supports local printing — that is, it will format and print output onto a printer that is plugged into a terminal's auxiliary port.

lp's commands resemble those used by UNIX System V to control printing, so this system can work more easily with third-party applications. Note, however, that the MLP implementation of **lp** does differ in some important respects from the System-V original; therefore, users who have used **lp** under UNIX should pay close attention to the following descriptions.

lp consists of the following commands:

cancel	Cancel the printing of a job
chreq	Change priority, lifetime, or printer for a job
lp	Spool one or more files for printing
lpadmin	Administer the print-spooler system
lpsched	Print jobs spooled with command lp; turn on printer daemon
lpshut	Stop the printer daemon
lpstat	Give the status of printer or print request
pclfont	Prepare a PCL font for downloading via MLP
reprint	Reprint a spooled print job
route	Let a user change his default printer

Each of these commands is described in its own Lexicon entry.

lp uses the following directories:

/usr/spool/mlp/backend	This directory holds the programs and scripts used to manage printers.
/usr/spool/mlp/queue	This directory holds all print requests.
/usr/spool/mlp/route	This directory holds files that name each user's default printer.

lp's behavior is set by the contents of the following files:

/usr/spool/mlp/controls	This file holds **lp**'s configuration data base. This data base links a printer by name to the device through which it is accessed, and to the configuration script (if any) with which its input is massaged. For information on how to modify it, see the Lexicon entry for **controls**.
/usr/spool/mlp/log	This file logs **lp**'s activity.
/usr/spool/mlp/status	This file gives the status of each defined printer.

To use **lp**, you must first use the command **lpadmin** to build a description file for each class of printer that you have plugged into your system. The description file names the class of printer (e.g., "epson" or "laserjet") and gives the information **lp** needs to manipulate input to the printer. For example, a script may include a **stty** command to set the port into a special mode, and one or more commands for filtering the input so it will print properly. A backend script can invoke commands like **prps** or **epson** to process text for printing. **lp** can perform sophisticated filtration; for example, it can correctly handle PostScript code that prints images or bar codes. See the Lexicon entry for **lpadmin** for more details on these scripts.

You must then use **lpadmin** to link a given printer, by name, to the device through which it is accessed. You must have first identified the port into which each printer is plugged, as described above. These links are stored in file **/usr/spool/mlp/controls**. If you have prepared a configuration script for this printer's type, then you must link it to the given printer as well. For example, if you have prepared a configuration script for all PostScript printers and named it **postscript**, then you must link that script to every PostScript printer whose input you want to be massaged in this manner. Unlike the **lpr** printing system, **lp** lets you attach to your computer more than one printer of each type.

One last point: each "printer" should identify a given physical device plus a given means of accessing it. Thus, one physical printer can have more than one name if you plan to access it in more than one manner. See the Lexicon entry for **lpadmin** for more information on this topic.

Note that if a printer is a "local printer" — that is, a printer plugged into the auxiliary port of the terminal that the user is using, the **termcap** description for that terminal must define the variables **PS** (print start) and **PN** (print end). Each printer's description file is stored in directory **/usr/spool/mlp/backend**.

You can use the command **route** to assign a default printer to each user. If the user has set a default printer for himself and if he does not name a printer on the **lp** command line, the output goes to that default printer. If the user has *not* set a default printer for himself and does not name a printer on his **lp** command line, the output goes to the system's default printer. This feature is an extension to the version of **lp** that is implemented by UNIX System V.

To spool a job for printing, use the command **lp**. A *job* consists either of one or more files, or of text read from the standard input. **lp** prefaces the job with a header that describes where and how the job is to be printed, then copies it into directory **/usr/spool/mlp/queue**. The name that **lp** gives the spooled job reflects its status, that is, the order in which it should be printed relative to other jobs that user has spooled. This allows each user to give a priority to the jobs that he has spooled.

Each job resides in the spooling directory until the printer daemon **lpsched** reads it and prints it. **lpsched** selects jobs for printing based on their relative priority, as shown in their names. It finds where the job is to be printed by reading its header; then it opens the description file for that printer and follows its directions for printing the job. To turn on the daemon, use the command **lpsched** by itself; to turn it off, use the command **lpshut**. If the spooler is shut down, jobs remain in **/usr/spool/mlp/queue** until you reawaken the daemon by issuing the command **lpsched**.

To see what files are being printed where, use the command **lpstat**. To cancel a printing request, use the command **cancel**.

A job remains "alive" in **/usr/spool/mlp/queue** until its "life" has expired; the life is set in its header. There are three types of "lifetime": *temporary*, in which a job survives two hours from the time of spooling; *short-term*, in which a job survives 48 hours; and *long-term*, in which a job survives 72 hours. The default is short-term. When a

job's life expires, **lpsched** removes it. A user can use the command **chreq** to change a job's lifetime or priority; or redirect it from one printer to another. While a job lives in the spool directory, a user can use the command **reprint** to reprint it. He can also use the command **route** to change his default printer.

Note that you should be *very* careful that jobs that include sensitive information — e.g., the payroll checks or your resume — do not linger in spool directory, where other users can reprint them. For information on resetting a job's lifetime, see the Lexicon entries for **chreq** and **MLP_LIFE**. You can change the default definitions of temporary, short-term, and long-term by editing **controls**. See its entry in the Lexicon for more information. *Caveat utilitor!*

The following environmental variables affect **lp**'s default behavior:

MLP_COPIES	The number of copies to print.
MLP_FORMLEN	The number of lines on the page to be printed.
MLP_LIFE	The "lifespan" of a spooled file.
MLP_PRIORITY	The default priority to give each spooled file.
MLP_SPOOL	Set a number of user-specific variable, such as title of document, type of document, and data base.

These variables can be set either by a user, or embedded in a script. Each is detailed in its own Lexicon entry.

See Also

Administering COHERENT, hpr, lp, lp [device driver], lpr, lpsched

Notes

When you link **/dev/lp** or **/dev/hp** to a device, it normally is linked to a "cooked" device, e.g., **/dev/hp**. This works correctly for character-based output, such as text (or PostScript files); however, if you are downloading binary data to the printer, such as graphics or fonts, be sure to use the "raw" device, e.g., **/dev/rhp**. Passing binary information through a "cooked" device will garble the data and distort the resulting image.

Some COHERENT 4.2 customers have experienced printing problems, including no printing, slow printing, or printing stops after a line or two. To fix this, one needs to do the following steps in exact order;

1. Edit file **/etc/conf/install_conf/keeplist**.
2. Change the last line so that it reads as follows:

   ```
   echo '-I SHMMNI:SEMMNI:NMSQID:LPWAIT:LPTIME:LPTEST'
   ```

3. Type the following command to build a new COHERENT kernel:

   ```
   /etc/conf/bin/idmkcoh -o /testcoh
   ```

4. Shutdown and reboot with the new kernel.
5. Log in as the superuser **root**.
6. Set the kernel variables that control discipline of the printer. The driver uses a hybrid busy-wait/timeout discipline, to efficiently support in a multi-tasking environment a variety of printers whose buffers come in a multiplicity of sizes.

   The variable **LPWAIT** sets the time for which the processor waits for the printer to accept the next character. If the printer is not ready within the **LPWAIT** period, the processor then resumes normal processing for the number of ticks set by the kernel variable **LPTIME**. Thus, setting **LPWAIT** to an extremely large number (e.g., 1,000) and **LPTIME** to a very small number (e.g., one) results in a fast printer, but leaves very few cpu cycles available for anything else. Conversely, setting **LPWAIT** to a small number (e.g., 50) and **LPTIME** to a large number (e.g., five) results in efficient multi-tasking but also results in a slow printer unless the printer itself contains a buffer (as is normal with all but the least expensive printers). By default, **LPWAIT** is set to 400 and **LPTIME** to four. We recommend that you set **LPWAIT** to no less than 50 and no more than 1,000 and **LPTIME** to no less than one.

   The variable **LPTEST** determines wether the device driver checks to see if the printer is in an "on-line" condition before it uses the device. If your printer does not support this signal, you must set **LPTEST** to zero.

   To reset the values of **LPWAIT**, **LPTIME**, and **LPTEST**, edit file **/etc/conf/mtune** and set the parameters **LPWAIT_SPEC**, **LPTIME_SPEC**, and **LPTEST_SPEC** to the values that you want. Then use the command **/etc/conf/bin/idmkcoh** to build a new kernel. For details on this command, see its entry in the Lexicon. One word of caution to the wary: be sure to name your new kernel something innocuous, such as **cohtest**, to ensure that you do not clobber your current working kernel.

7. Reboot the new kernel and try printing again.

8. If your printer still exhibits problems, try increasing or decreasing the values of **LPTIME** and **LPWAIT**. Remember, each time you build a new kernel kernel, you must reboot in order for the new variables to take effect.

The MLP printer spooler is distributed under license from Magnetic Data Operations, 9400B Two Notch Road, Columbia, SC 29223.

The message

```
cannot open device /dev/lp
```

from **lpr** means either that the printer is not turned on, or that the device **/dev/lp** is not linked to the correct parallel-port device. Use the directions given above to find and link the correct device. The same applies when you receive this message from **hpr**.

printf() — STDIO Function (libc)

Print formatted text
#include <stdio.h>
int printf(*format* **[,***arg1, ... argN***])**
char **format***format***; [data type]** *arg1, ... argN***;**

printf() prints formatted text. It uses the *format* string to specify an output format for each *arg*, which it then writes on the standard output.

printf() reads characters from *format* one at a time; any character other than a percent sign '%' or a string that is introduced with a percent sign is copied directly to the output. A '%' tells **printf()** that what follows specifies how the corresponding *arg* is to be formatted; the characters that follow '%' can set the output width and the type of conversion desired. The following modifiers, in this order, may precede the conversion type:

1. A minus sign '-' left-justifies the output field, instead of the default right justify.

2. A string of digits gives the *width* of the output field. Normally, **printf()** pads the field with spaces to the field width; it is padded on the left unless left justification is specified with a '**-**'.

   If the field width begins with '0', the field is padded with '0' characters instead of spaces; the '0' does not cause the field width to be taken as an octal number. Note that this applies only to numeric string descriptors. If the field descriptor describes a character or string (i.e., **%c** or **%s**), **printf()** ignores a leading '0' and always pads the field with spaces.

   If the width specification is an asterisk '*', the routine uses the next *arg* as an integer that gives the width of the field.

3. A period '**.**' followed by one or more digits gives the *precision*. For floating point (**e**, **f**, and **g**) conversions, the precision sets the number of digits printed after the decimal point. For string (**s**) conversions, the precision sets the maximum number of characters that can be used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer that gives the precision.

4. The letter '**l**' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect; note, however, that capitalized conversion types are *not* compatible with all C compiler libraries, or with the ANSI standard. This feature will not be supported in future editions of COHERENT.

The following format conversions are recognized:

% Print a '%' character. No arguments are processed.

c Print the **int** argument as a character.

d Print the **int** argument as signed decimal numerals.

e Print the **float** or **double** argument in exponential form. The format is *d.ddddddesdd*, where there is always one digit before the decimal point and as many as the *precision* digits after it (default, six). The exponent sign *s* may be either '+' or '-'.

f Print the **float** or **double** argument as a string with an optional leading minus sign '-', at least one decimal digit, a decimal point ('**.**'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default, six).

g Print the **float** or **double** argument as whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.

ld Print the **long** argument as signed decimal numerals.

lo Print the **long** argument in unsigned octal numerals.

lu Print the **long** argument in unsigned decimal numerals.

lx Print the **long** argument in unsigned hexadecimal numerals.

o Print the **int** argument in unsigned octal numerals.

p The ANSI standard states that the behavior of the **%p** descriptor is implementation-specific. Under COHERENT, **%p** prints in format **%#.8X** the literal value of a pointer. Its corresponding variable must be of type **char** *.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

This descriptor is not part of the ANSI Standard. Its use is deprecated. Code that uses it may not be portable to other systems.

s Print the string to which the **char** * argument points. Reaching either the end of the string, indicated by a null character, or the specified *precision*, will terminate output. If no *precision* is given, only the end of the string will terminate.

u Print the **int** argument in unsigned decimal numerals.

x Print the **int** argument in unsigned hexadecimal numerals. The digits are prefaced by the string **0x**.

X Like **%x**, except that the digits are prefaced by the string **0X**. Note COHERENT release 4.2 has changed the means of **%X** to conform to the ANSI C standard. In versions prior to release 4.2, this format conversion printed a **long** argument in unsigned hexadecimal numerals. Programs that depend upon the obsolete use of **%X** will no long work the same under the current release of COHERENT.

If it wrote the formatted string correctly, **printf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

This example implements a mini-interpreter for **printf()** statements. It is a convenient tool for seeing exactly how some of the **printf()** options work. To use it, type a **printf()** conversion specification at the prompt. The formatted string will then appear. To reuse a format identifier, simply type **<return>**:

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* the replies go here */
static char reply[80];

/* ask for a string and echo it in reply. */
char *askstr(msg)
char *msg;
{
      printf("Enter %s ", msg);
      fflush(stdout);

      if (gets(reply) == NULL)
            exit(EXIT_SUCCESS);
      return (reply);
}

main()
{
      char fid[80], c;
```

```
      /* initialize to an invalid format identifier */
      strcpy(fid, "%Z");

      for (;;) {
            askstr("format identifier");
            /* null reply uses previous FID */
            if (reply[0])
                  /* leave the '%' */
                  strcpy(fid + 1, reply);

            switch(c = fid[strlen(fid) - 1]) {
            case 'd':
            case 'i':
                  askstr("signed number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, atoi(reply));
                  break;

            case 'o':
            case 'u':
            case 'x':
            case 'X':
                  askstr("unsigned number");
                  if(strchr(fid, 'l') != NULL)
                        printf(fid, atol(reply));
                  else
                        printf(fid, (unsigned)atol(reply));
                  break;

            case 'f':
            case 'e':
            case 'E':
            case 'g':
            case 'G':
                  printf(fid, atof(askstr("real number")));
                  break;

            case 's':
                  printf(fid, askstr("string"));
                  break;

            case 'c':
                  printf(fid, *askstr("single character"));
                  break;

            case '%':
                  printf(fid);
                  break;

            case 'p':
                  /* print pointer to format id */
                  printf(fid, fid);
                  break;

            case 'n':
                  printf("n not implemented");
                  break;

            default:
                  printf("%c not valid", c);
            }

            printf("\n");
      }
}
```

See Also

ecvt(), fcvt(), fprintf(), gcvt(), libc, putc(), puts(), scanf(), sprintf(), vprintf()
ANSI Standard, §7.9.6.3

POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that each argument match its counterpart in the format string.

Versions of COHERENT prior to release 4.2 recognized the conversion formats **%D**, **%O**, and **%U**. The ANSI standard does not recognize these conversion characters, and beginning with release 4.2 the COHERENT implementation of **printf()** no longer recognizes them. You should instead use, respectively, the conversion characters **%ld**, **%lo**, and **%lu**.

proc.h — Header File

Define structures/constants used with processes
#include <sys/proc.h>

proc.h defines structures and constants used by routines that manipulate processes.

See Also

header files

process — Definition

A **process** is a program in the state of execution.

See Also

daemon, file, Using COHERENT

prof — Command

Print execution profile of a C program
prof [-abcs][*progfile* **[** *monfile* **]]**

prof interprets the profile file produced by an execution of a C program and reports the execution frequencies of each routine. It also reports the percentage of execution time spent in each routine.

prof normally reports times and frequencies spent for regions of programs between externally defined names. *progfile* is the executable program; if omitted, **a.out** is assumed. *monfile* is the monitor file produced during execution of the program; if omitted, **mon.out** is assumed.

To produce **mon.out**, a program must be compiled with the **-VPROF** option to **cc**. To profile all modules, each module must be compiled with this option.

The following options are available.

-a Profile all symbols, not just externals.

-b Print all bin information.

-c Print all call information.

-s Report stack usage high-water mark.

Files

a.out — Program file (with name list intact)
mon.out — Raw execution profile

See Also

cc, commands, ld, nm

profile — System Administration

Set default environment at login
/etc/profile

The shell executes the script **/etc/profile** whenever any user logs in. This script sets up the default environment for a user. Note that the actions of this script can be altered or supplemented by each user's **.profile** script.

If **/etc/passwd** specifies a program in the login-shell slot, then **/etc/profile** is read by **/bin/sh**. Those lines that begin with the command **export** are recognized as global environments, and the remainder of the line is inserted

into the environment.

Please note that if **/bin/sh** or **/bin/ksh** is not the shell, any constructions other than

```
export foo=value
```

are not likely to work.

See Also

Administering COHERENT, ksh, .kshrc, .profile, sh

Programming COHERENT — Overview

The C language is the "native language" of COHERENT. Most COHERENT programs are written in C.

If you are a beginner and are interested in learning something about C, look at the tutorial *The C Language* in the first part of this manual.

The following Lexicon entries give you information you need to write or port C programs under COHERENT:

C keywords
This lists the C keywords recognized by the COHERENT implementation of C. Each keyword, in turn, is described in full in its own Lexicon entry.

C language
This summarizes the COHERENT implementation of C. It gives the size of each data type, formatting of floating-point data, static limits, and other information.

C preprocessor
This describes the processing directives that the COHERENT preprocessor recognizes. Each directive is described in full in its own Lexicon entry

header files
This entry names the header files included as part of COHERENT. Each header file is described in its own Lexicon entry. Some of the header-file articles are of particular interest.

libraries
This describes the libraries included with COHERENT. Almost every library function and system call has its own Lexicon entry; the only exceptions are the routines kept in **libmisc.a** and **libcurses.a**. Each library has its own summary entry; of particular interest are the entries **libc**, **libm**, **libgdbm**, and **libsocket**.

If you are an experienced C programmer who is new to COHERENT, we suggest you look first at the article for **C language**, to get an overview of the dialect of C that COHERENT supports. Look at the entry for libraries, to see what libraries are available; then look at the entry for each library to see what functions are available.

The following Lexicon entries describe the commands with which you can compile and manage your programs:

ar The archiver. This turns a group of object modules into a library.

as The COHERENT macro-assembler. This assembles modules written in assembly language, and builds object modules that you can link with modules written in C or other languages.

cc The C compiler. This describes the compiler itself, and its options and switches.

cpp The C preprocessor. The preprocessor itself has its own options to help you control the building of your programs.

db The symbolic debugger. With **db**, you can set breakpoints, single-step through code, hot-patch binaries, and otherwise debug your programs. It requires knowledge of 80386 assembly language.

ld The linker. This links object modules into an executable binary. The Lexicon entry describes its switches and features.

make The programming discipline. **make** helps you to manage the building of a complex program. It is indespensible for managing all but the simplest programming projects.

nm This utility prints the contents of a program's symbol table.

sh The Bourne shell. This is of the COHERENT command interpreter. You can write large, complex programs in the shell. These can functions, and draw on a library of prewritten functions. The shell is one of the most powerful tools available to a COHERENT programmer — and one of the most neglected.

strip Strip the symbol table from a program. This makes most programs significantly smaller, with no loss in functionality.

Each command is described in its own Lexicon entry.

Definitions

The following Lexicon entries give technical definitions of interest to programmers:

address
What an "address" is.

alignment
What byte alignment is, and how it applies under the various machine on which COHERENT has been implemented

ANSI A brief introduction to the ANSI Standard for Programming Language C.

arena What an arena is, and how it applies to COHERENT programs.

array What an array is, and elementary information on how to code it.

ASCII The ASCII table.

bit What a bit is.

bit map
What a bit map is, and how to code it under C.

buffer What a buffer is, and how buffering affects your languages.

byte What a byte is.

byte ordering
This describes how bytes and words are ordered on the various machines on which COHERENT has been implemented.

calling conventions
The calling conventions for COHERENT functions. This is particularly important if you are writing modules in assembly language.

cast How to "coerce" one data type into another.

cc0 The COHERENT C parser.

cc1 The COHERENT C code generator.

cc2 The COHERENT C optimizer.

cc3 The COHERENT de-compiler. It generates a file of assembly language for your examination.

data formats
This gives the size of the common data types on the various machines on which COHERENT has been implemented.

data types
The data types that COHERENT C recognizes.

environ
This article introduces the argument **environ**, which by default is the third argument passed to the function **main()** in a C program. It points to image of the process's environment.

errno This global variable holds the error status returned by a COHERENT system call. The article **errno.h** interprets the codes that can appear in this variable.

execution
This describes how each form of the system call **exec()** executes a program.

field Description of what a field is, and how to address it.

FILE Description of the **FILE** structure used by STDIO routines.

file What a file is. It also goes into the "black art" of permissions.

file descriptor
Description of the file descriptor used by COHERENT system calls.

function
What a function is.

GMT A brief introduction to Greenwich Mean Time, which is the internal time for every COHERENT system.

initialization
This describes the rules of initialization for C.

interrupt
What an interrupt is.

Latin 1 The table ISO Latin 1 (ISO 8859.1).

lvalue Definition of the "left value" in a C expression.

macro What a C macro is, and how COHERENT C processes them.

manifest constant
This introduces manifest constants, and lists the constants that COHERENT defines automatically.

modulus
A definition of the modulus arithmetic operation.

NUL Definition of the NUL character.

nybble What a "nybble" is.

object format
Definition of an object format.

operator
A list of the C operators. This article also gives a table of precedence for the operators.

pattern
What a pattern is.

pointer
What a pointer is, and tips for using pointers with COHERENT C.

portability
This gives some tips on how to write portable programs.

POSIX Standard
A brief introduction to the POSIX Standard

random access
A definition of random access.

read-only memory
A definition of ROM, or "read-only memory".

recursion
A definition of this programming technique.

rvalue Definition of the "right value" in a C expression.

signame
This global array holds a string that describes the signal that a program has received.

stack A definition of the program stack, and how to manipulate it under COHERENT C.

standard error
Definition of the standard-error device.

standard input
Definition of the standard-input device.

standard output
Definition of the standard-output device.

stderr The file descriptor of the standard-error device.

stdin The file descriptor of the standard-input device.

STDIO Definition of STDIO — i.e., "standard input and output".

stdout The file descriptor of the standard-output device.

storage class
This entry summarizes the classes of storage that COHERENT C recognizes.

stream Definition of a file stream.

STREAMS
This article summarizes the COHERENT implemenation of STREAMS.

structure
Definition of a structure, and basic information on how to code it.

structure assignment
This details structure assigment under COHERENT C.

stty Summary of the **stty** interface to terminals.

termio Introduction to the **termio** terminal interface.

termios
This summarizes the POSIX Standard extensions to the **termio** terminal interface.

type checking
This details type checking under COHERENT C.

type promotion
This details type promotion under COHERENT C.

Other Languages

COHERENT includes the following programming languages:

awk This interpreted language lets you write programs for text processing. It is especially good at processing tabular information, thus letting you quickly write simple data-base programs.

bc **bc** is a calculator program that offers infinite magnitude and infinite precision. This is an interpreted langauge that you can program on the fly to perform simple tasks, such as computing interest payments on the national debt. You can also write programs that you can run repeatedly. These can also take advantage of a library of routines already written for you.

lex This program reads a set of lexical analysis rules that you write in a standard form, and generates a C program that you can compile and run.

yacc This program reads a set of parsing rules that you write in Backus-Naur Form, and generates a C program that you can compile and run. You can use with code generated by **lex** to write complex programs, such as compilers.

Each of these languages is described in a Lexicon article. The front of the manual has a tutorial for each.

See Also

Administering COHERENT, C language, COHERENT, commands, libraries, Using COHERENT

protocols — System Administration

Name communications protocols
/etc/protocols

The file **/etc/protocols** describes the Internet protocols that your local host recognizes. Each line within this file describes one protocol. A description consists of the following fields:

- The protocol's official name.
- Its number.
- Aliases, if any, for the protocol name.

Any text that follows the character '#' is comment, and is ignored by any program that reads this file.

For example:

```
icmp    1    ICMP    # internet control message protocol
ggp     3    GGP     # gateway-gateway protocol
tcp     6    TCP     # transmission control protocol
```

See Also

Administering COHERENT, hosts, hosts.equiv, inetd.conf, networks, services

prps — Command

Prepare files for PostScript-compatible printer
prps [*options***] [***file ...* **]**

prps reads each *file*, breaks it into pages, writes a header at the top of each page, then writes the paginated text onto the standard output. If no *file* is given, **prps** reads the standard input.

Unike the related command **pr**, **prps** writes its output in the PostScript language, suitable for printing on a PostScript printer such as an Apple LaserWriter or a Hewlett-Packard LaserJet with a PostScript cartridge. The PostScript output program generates a sequence of standard 8.5×11-inch pages, each containing a header line (file name, current time and date, and page number) and a box that encloses the text of *file*. The default output typeface is ten-point Courier.

prps recognizes the following options:

-b Suppress the box around the page text. If the box is present, PostScript clips text that would extend beyond its right border.

-h Suppress the header line.

-i*n* Indent the left margin by an additional *n* characters.

-l Generate "landscape"-format output. **prps** normally generates output pages in "portrait" format (upright 8.5×11 inches). The **-l** option generates output pages in landscape format (11 by 8.5) instead. This option is useful for files with long lines; by default, it prints 46 lines per page.

-l2 Generate landscape-format output pages that each contain two side-by-side "pages" of text. This format is useful for saving paper, especially when used with a small size of type. As it prints in a small size of type, it prints 66 lines per page.

-n*name* Use *name* in place of the file name in the header line.

-t*N* Set tab stops at every *N* characters. The default tab setting is eight.

-ptsize Change the size of type to *ptsize* points. By default, **prps** sets its output in ten-point type. This yields 64 lines per normal output page, 46 lines in landscape format, and 52 lines per half page in **-l2** format. (Note that a "point" is one twelfth of a pica, which in turn is one sixth of an inch; thus, there are 72 points in an inch.) By specifying the *ptsize* on its command line, you can tell **prps** to use a different size of type. For example, **-8** tells **prps** to use eight-point type.

-p*N* Print *N* lines of text on each output page (or half page). Note that the point size determines how many lines fit on a page, and lines per page determine point size. If you specify both, **prps** will use the given values unless the lines do not fit at the given point size.

+*N* Skip the first *N* output pages.

Setting Fonts

prps recognizes the standard **nroff** font specification sequences and translates them into PostScript font specifications. The default font is Courier. Because the naming conventions for PostScipt fonts are anything but uniform, **prps** appends a suffix to the fontname to designate a Roman, boldface and italic font variety. The default suffix is ' ' for Roman, "-Bold" for bold and "-Oblique" for italic. These give the standard PostScript names for the Courier family, "Courier", "Courier-Bold", and "Courier-Oblique".

Option **-f***fontname* specifies an alternative *fontname.* Option **-Fs***Xsuffix* specifies an alternative font suffix, where *X* is one of the three characters **RBI** (for **R**oman, **B**old or **I**talic) and *suffix* is the desired suffix. For example, the option

```
-fTimes -FsR-Roman -FsI-Italic
```

generates the usual PostScript font names for the Times family, namely "Times-Roman", "Times-Bold", and "Times-Italic".

To spare you some of this grief, a few fonts have built-in abbreviations. Option **-F***X*, where *X* is one of the characters **ABHNPST,** specifies a PostScript fontname as follows:

-FA AvantGarde
-FB Bookman
-FH Helvetica
-FN Helvetica-Narrow
-FP Palatino
-FS New Century Schoolbook
-FT Times

These options also set each suffix appropriately for the desired font. However, font naming conventions may differ on various PostScript devices; examine the **prps** output and your device documentation if problems occur.

Examples

prps is especially useful as a way of printing the output of **nroff**, including manual pages. For example,

```
man prps | prps | hpr -B
```

or

```
man prps | prps -l2 | hpr -B
```

prints this Lexicon article in, respectively, portrait mode or two-page landscape mode. It looks nicer if you center the output with an indent:

```
man prps | prps -i8 | hpr -B
```

or

```
man prps | prps -l2 -i4 | hpr -B
```

See Also

commands, hp, hpr, lp, pr, nroff, printer

Notes

When you installed COHERENT onto your system, the installation program asked you whether your printer used the PostScript language. For information on how to install a PostScript printer onto your system, see the Lexicon entries for **lp** and **printer**.

ps — Command

Print process status
ps [-][adefglmnrtwx] [-c *sys***] [***mem***] [-p***pid,pid,...,pid]*

ps prints information about a process or processes. It prints the information in fields, followed by the command name and arguments. The fields include the following:

TTY The controlling terminal of the command, printed in short form. For example, "tty44:" means **/dev/tty44**. A dash means there is no controlling terminal.

PID Process id; necessary to know when the process is to be killed.

GROUP PID of the group leader of the process, that is, the shell that started up when the user logged in.

PPID PID of the parent of the process; very often a shell.

UID User id or name of the owner.

K Size of the process, in kilobytes.

F Process flag bits, as follows:

PFCORE	00001	Process is in core
PFLOCK	00002	Process is locked in core
PFSWIO	00004	Swap I/O in progress
PFSWAP	00010	Process is swapped out
PFWAIT	00020	Process is stopped (not waited)
PFSTOP	00040	Process is stopped (waited on)
PFTRAC	00100	Process is being traced
PFKERN	00200	Kernel process
PFAUXM	00400	Auxiliary segments in memory
PFDISP	01000	Dispatch at earliest convenience
PFNDMP	02000	Command mode forbids dump
PFWAKE	04000	Wakeup requested

S State of the process, as follows:

R	Ready to run (waiting for CPU time)
S	Stopped for other reasons (I/O completion, pause, etc.)
T	Being traced by another process
W	Waiting for an existent child
Z	Zombie (dead, but parent not waiting)

EVENT The condition that the process is anticipating. This not applicable if the process is ready to run. The following gives the legal symbolic names of events. If a driver does not support symbolic event names, **ps** prints a unique hexadecimal number instead:

System Sleeps:

bpwait	Wait for a buffer to become valid
bufneed	Wait for a free buffer to become available
bwrite	Wait for a buffer write to finish
ioreq	An IO request is being processed
pause	This process is in the **pause()** system call
pipe data	Wait for data to appear in a pipe
pipe wx	
poll	Wake for polled event, poll timeout, or signal
ptrace	Send a **ptrace** command to a traced child
ptret	Wait for signal processing in a traced child to complete
pwrite	Wait for a pipe to empty enough for a write
swap	Wait for a process to get swapped in
wait	Wait for a child to terminate
waitq	Wait for more character queues to become available

Driver Sleeps

aha:ccb	AHA-154x driver is waiting for a SCSI command to complete
nkbcmd	
nkbcmd...	
nkbcmd2	
nkbcmd2...	**nkb** is waiting for a command to complete
ptycd	Pseudoterminal driver is waiting for carrier
ptyread	Pseudoterminal driver is waiting for a read
ptywrite	Pseudoterminal driver is waiting for a write
ttydrain	Line discipline is waiting for a tty to drain
ttyiodrn	**ioctl()** asked line discipline to let tty output drain
ttyoq	Line discipline is waiting for an output queue to drain
ttywait	Line discipline is waiting for more data

CVAL SVAL IVAL RVAL
Scheduling information; bigger is better.

UTIME Time consumed while running in the program (in seconds).

STIME Time consumed while running in the system (in seconds).

Normally, **ps** displays the **TTY** and **PID** fields of each active process started on the caller's terminal, as well as the command name and arguments. The following flags alter this behavior.

-a Display information about processes started from all terminals.

-c *sys* This option does nothing; it is included to preserve the integrity of some shell scripts.

-d Print information about status of loadable drivers.

-e Same as **-a**. This is included for compatibility with other implementation of **ps**.

-f Blank fields have '-' place-holders. This enables field-oriented commands like **sort** and **awk** to process the output.

-g Print the group leader field GROUP if the **l** option is given.

-k *mem* The next argument *mem* is the memory image (default, **/dev/mem**). Note that this argument currently does nothing; it is included only to preserve old shell scripts. The COHERENT implementation of **ps** reads information from **/dev/ps**. This permits **ps** to be smaller and faster, helps to avoid "ghosts," and to be atomic.

-l Long format. In addition to the TTY and PID fields, prints the PPID, UID, K, F, S and EVENT fields.

-m This option does nothing; it is included to preserve the integrity of some shell scripts.

-n Suppress the header line.

-p*pid,pid,...,pid*
Print information for each process identifier *pid* in the comma-separated list.

-r Print the real size of the process, which includes the user and auxiliary segments assigned to the process. Because the user segment (usually 1 kilobyte) is shared by all processes owned by that user, this may give a misleading total size for all the user's processes.

-t Print elapsed CPU time fields UTIME and STIME.

-w Wide format output; print 132 columns instead of 80.

-x Display processes which do not have a controlling terminal.

Files

/dev/ps — Device for a system driver
/dev/tty* — List of terminal names

See Also

commands, hmon, kill, mem, ps [device driver]**, size, wait**

Notes

Each process can modify or destroy its command name and arguments. The state of the system changes even as **ps** runs.

ps — Device Driver

Driver to return information about processes
/dev/ps

The file **/dev/ps** accesses the kernel's process table. It is a part of the driver **mem**, which manages memory; thus, it has major number 0 and minor number 6.

/dev/ps is a read-only device that exists only to support the command **ps** and its variants. The command **ps** reads this device to display a "snapshot" of the processes that the COHERENT kernel is executing.

Reading **/dev/ps** deposits an array of the structure **stMonitor** into the read buffer. The number of bytes requested by the system call **read()** should be enough to accommodate the entire process table. Header file **<sys/coh_ps.h>** defines **stMonitor**.

See Also

device drivers, ps [command]

PS1 — Environmental Variable

User's default prompt
PS1=*prompt*

The environmental variable **PS1** sets the prompt for your shell. The default is **$**.

See Also

environmental variables, PS2, sh

PS2 — Environmental Variable

Prompt when user continues command onto additional lines
PS2=*prompt*

The environmental variable **PS2** sets the prompt that is displayed when a command extends onto additional input lines. The default is **>**.

See Also

environmental variables, PS1, sh

PSfont — Command

Cook an Adobe font into PostScript format
PSfont [-qs] [*infile***.pfb [** *outfile* **]]**

The command **PSfont** "cooks" a file that is in Adobe's downloadable-font format into PostScript. The output of **PSfont** can either be loaded into your PostScript printer as a memory-resident font, which can be used across multiple files, or included within the output of **troff**.

PSfont recognizes two options:

-q Quiet option: suppress the printing of warning messages. **PSfont** normally complains about error conditions it finds within fonts, such as extraneous control characters.

-s Suppress the instructions **serverdict** and **exitserver** from the output. Use this option if you wish to include the output of **PSfont** within **troff** output; do *not* use this option if you want the cooked font to be resident within the printer after you download it.

infile is the Adobe font file that **PSfont** cooks into PostScript. It must have the suffix **.pfb**. If you do not name an *infile* on the command line, **PSfont** reads the standard input.

outfile names the file into which **PSfont** writes its output. By convention, it should have the suffix **.ps**, although this is not required. If you do not name an *outfile* on the command line, **PSfont** writes to the standard output.

See Also

commands, fwtable, troff
Supporting downloadable PostScript language fonts, Adobe Technical Note No. 5040, §3.3. Mountain View, Ca., Adobe, Incorporated, 1992.

Notes

For more information on using **PSfont** with **troff**, see the Lexicon entry for **troff**.

ptrace() — System Call (libc)

Trace process execution
#include <signal.h>
int ptrace(*command, pid, location, value***)**
int *command, pid*, ******location*, *value*;

ptrace() provides a parent process with primitives to monitor and alter the execution of a child process. These primitives typically are used by a debugger such as **db**, which needs to examine and change memory, plant breakpoints, and single-step the child process being debugged.

Once a child process indicates it wishes to be traced, its parent issues various *command*s to control the child. *pid* identifies the affected process. The parent may issue a command only when the child process is in a stopped state, which occurs when the child encounters a signal. A special return value of 0177 from **wait()** informs the parent that the child has entered the stopped state. The parent may then examine or change the child process memory space or restart the process at any point.

When the child process issues an **exec()**, the child stops with signal **SIGTRAP** to enable the parent to plant breakpoints. The set user id and set group id modes are ineffective when a traced process performs an **exec()**.

The following list describes each available *command*. A *command* ignores any arguments not mentioned.

0 This is the only *command* the child process may issue. It tells the system that the child wishes to be traced. Parent and child must agree that tracing should occur to achieve the desired effect. Only the *command* argument is significant.

1,2 The **int** at *location* is the return value. Command 1 signifies that *location* is in the instruction space, whereas command 2 signifies *data* space. Often these two spaces are equivalent.

3 The return value is the **int** of the process description, as defined in **sys/uproc.h.** This call may be used to obtain values such as hardware register contents and segment allocation information.

4,5 Modify the child process's memory by changing the **int** at *location* to *value.* Command 4 means instruction space and command 5 means data space. Shared segments may be written only if no other executing process is using them.

6 Modify the **int** at *location* in the process description area, as with command 3. The permissible values for *location* are restricted to such things as hardware registers and bits of machine status registers that the user may safely change.

7 This command restarts the stopped child process after it encounters a signal. The process resumes execution at *location,* or from where the process was stopped if *location* is **(int *)1.** *value* gives a signal number that the process receives as it restarts. This is normally the number of the signal that caused the process to stop, fetched from the process description area by a **3** command. If *value* is zero, the effect of the signal is ignored.

8 Force the child process to exit.

9 Like command **7**, except that the child stops again with signal **SIGTRAP** as soon as practicable after the execution of at least one instruction. The actual hardware method used to implement this command varies from machine to machine, explaining the imprecise nature of its definition. This call may provide part of the basis for breakpoints.

Files

<signal.h>
<sys/uproc.h>

See Also

db, exec, libc, ptrace.h, signal(), wait()

Diagnostics

ptrace() returns -1 if *pid* is not the process id of an eligible child process or if some other argument is invalid or out of bounds. Some commands may return an arbitrary data value, in which case **errno** should be checked to distinguish a return value of -1 from an error return.

Notes

There is no way to specify which signals should not stop the process.

ptrace.h — Header File

Perform process tracing
#include <sys/ptrace.h>

The header file **ptrace.h** holds definitions used by routines that perform process tracing. Among other things, it defines the structure **ptrace**.

See Also

header files

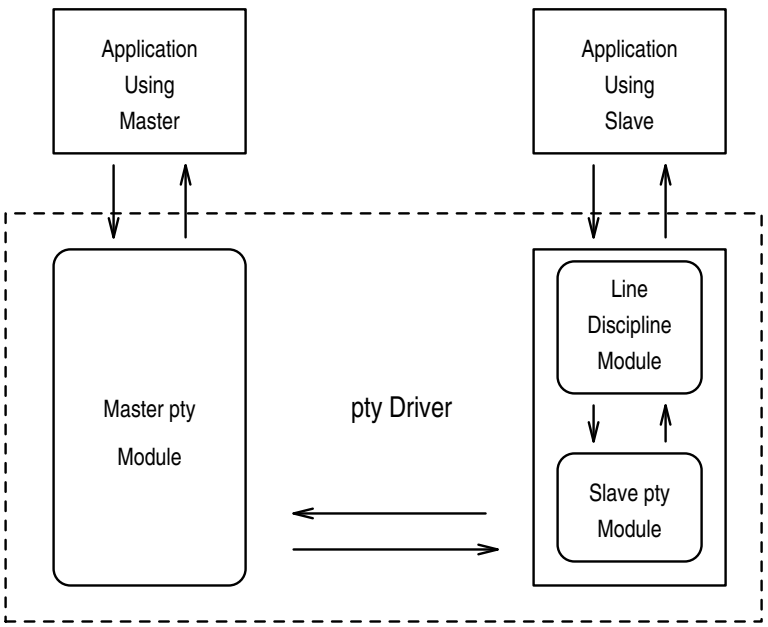
pty — Device Driver

Device driver for pseudoterminals

The COHERENT device driver **pty** lets your system support up to 128 pairs of pseudoterminals, or *ptys*.

A *pseudoterminal* is a means of letting a process masquerade as a terminal. For example, when you run the program **xterm** under X, that program passes what you type into COHERENT through a pseudoterminal device.

Each pseudoterminal consists of a pair of devices: a master device and a slave device. The program that is accepting input from a human at a keyboard (e.g., **xterm**) is "plugged" into the slave device; the program that is accepting and processing the input (e.g., a shell) is plugged into the master device. The following diagram shows how this pair of devices relate to each other:



As you can see, the slave device talks to the keyboard through a sub-module that performs line discipline. *Line-discipline* handles backspace characters, handles special interrupt characters (such as **<ctrl-C>**), and converts line-feed characters into carriage-return—line-feed character pairs: it bundles what you type into a package that can be passed to the master application and processed.

Only one process at a time can open a master device; the device is opened as soon as requested. Several processes can open a slave device, but blocks until the matching master device has been open. When blocked in this way, the slave is said to be "waiting for pseudocarrier."

An attempt to read a master device when no input is available, or to write to a master device when the slave cannot accept data, will block unless nonblocking I/O has been specifically requested; in this case, the system calls **read()** or **write()** fail and **errno** is set to EAGAIN.

You can use the system call **ioctl()** on slave devices with all valid line-discipline commands, including **TCGETA**, **TCSETA**, **TCSETAW**, **TCSETAF**, and **TCFLSH**. There are no valid **ioctl()** commands for master devices.

The system call **poll()** is allowed with both master and slave **pty** devices. However, priority polls (**POLLPRI**) are not supported.

Master devices are named **/dev/pty[p-w][0-f]**. Corresponding slaves are **/dev/tty[p-w][0-f]**. Like any other device, each **pty** has a major and minor number. The major number is 9 (**PTY_MAJOR** in system header file **<sys/devices.h>**). For slave devices, minor numbers are assigned according to the following scheme:

device	*Major number*	*Minor number*
/dev/ttyp0	9	0
/dev/ttyp1	9	1
...		
/dev/ttyp9	9	9
/dev/ttypa	9	10
/dev/ttypb	9	11
...		
/dev/ttypf	9	15
/dev/ttyq0	9	16
...		
/dev/ttyw0	9	112
...		
/dev/ttywf	9	127

For master devices, use **pty** instead of **tty** in the device name, and add 128 to the minor number.

The configurable parameter **NUPTY_SPEC** sets the number of **pty** pairs that may be used. The default is eight. If you want to change this value, invoke the script **/etc/conf/pty/mkdev** and enter the new value at the appropriate prompt. Then use the command **/etc/conf/bin/idmkcoh** to build a new kernel that incorporates this change; when the new kernel is built, boot it. For details, see the Lexicon entry for the command **idmkcoh**.

Specifying a value of zero for **NUPTY_SPEC** will cause the **pty** device to be omitted from the next kernel that **idmkcoh** generates.

See Also

device drivers

pushd — Command

Push an item onto the directory stack
pushd *[directory0 ... directoryN]*

The COHERENT shell **sh** maintains an internal "directory stack", which is a stack of names of directories. You can manipulate this stack should you, for any reason, wish to traverse a number of directories quickly and efficiently.

The command **pushd** pushes *directory1* through *directoryN* onto the directory stack, and changes the current directory to the last directory pushed. If called without an argument, it transposes the last two directories on the directory stack.

See Also

commands, dirs, popd, sh

putc() — STDIO Function (libc)

Write character into stream
#include <stdio.h>
int putc(*c***,** *fp***) char** *c***; FILE** **fp***;*

putc() writes character *c* into the file stream to which *fp* points. It returns *c* upon success.

Example

The following example demonstrates **putc()**. It opens an ASCII file and prints its contents on the screen. For another example of **putc()**, see the entry for **getc()**.

```
#include <stdio.h>
main()
{
      FILE *fp;
      int ch;
      int filename[20];

      printf("Enter file name: ");
      gets(filename);

      if ((fp = fopen(filename,"r")) != NULL) {
            while ((ch = fgetc(fp)) != EOF)
                  putc(ch, stdout);
      } else
            printf("Cannot open %s.\n", filename);
      fclose(fp);
}
```

See Also

fputc(), getc(), libc, putchar()
ANSI Standard, §7.9.7.8
POSIX Standard, §8.1

Diagnostics

putc() returns **EOF** when a write error occurs.

Notes

Because **putc()** is a macro, arguments with side effects may not work as expected.

putchar() — STDIO Function (libc)

Write a character onto the standard output
#include <stdio.h>
int putchar(*c***)**
char *c***;**

putchar() is a macro that expands to **putc(***c***, stdout)**. It writes a character onto the standard output.

Example

For an example of this routine, see the entry for **getchar()**.

See Also

fputc(), libc, putc()
ANSI Standard, §7.9.7.9
POSIX Standard, §8.1

Diagnostics

putchar() returns **EOF** when a write error occurs.

Notes

Because **putchar()** is a macro, arguments with side effects may not work as expected.

putenv() — General Function (libc)

Add a string to the environment
#include <stdlib.h>
int putenv (*envstring***)**
char **envstring**;

The function **putenv()** puts *envstring* into the user's environment. You can use this function to set a new environmental variable, or to change the definition of an existing variable.

envstring must point to a string of the form *VARIABLE*=*value*, where *VARIABLE* is the environmental variable being set, and *value* is the value to which it is being set.

putenv() returns zero if all goes well. If something goes wrong, it returns a value other than zero.

See Also

environ, environmental variables, getenv(), libc, stdlib.h

Notes

The global variable **environ**, which points to a process's environment, points to an array of pointers to strings rather than to an array of strings. When **putenv()** inserts *envstring* into the environment, it calls **malloc()** to enlarge the array of string pointers to which **environ** points, then inserts a pointer to *envstring* into that array. It does not copy *envstring* anywhere.

If a process uses **putenv()** to insert a string pointer into the environment, it can also call **getenv()** to read back that string; however, the array of strings passed to the process via **envp** (the third argument to the function **main()**) is not affected by a call to **putenv()**. For details on **environ** and **envp**, see their entries in the Lexicon.

It is an error to call **putenv()** with a pointer to an automatic variable as the argument, and then exit the calling function while *envstring* is still part of the environment. For safety's sake, *envstring* should point to a string that is static or global. See the Lexicon entry for **static**, or see the ANSI Standard §3.5.1.

putmsg() — System Call (libc)

Place a message onto a stream
#include <stropts.h>
int putmsg (*fd*, *ctlptr*, *dataptr*, *flags*)
int *fd*, *flags*; **const struct strbuf** **ctlptr*, **dataptr*;

putmsg() creates a message from user-specified buffer (or buffers), and sends the message to a STREAMS file. The message can contain either a data part, a control part, or both. The data and control parts to be sent are distinguished by being placed in separate buffers, as described below. The semantics of each part are defined by the STREAMS module that receives the message.

fd gives a file descriptor that identifies an open stream. *ctlptr* and *dataptr* each point to a structure of tyupe **strbuf**, which contains the following members:

```
int len;    /* Length of data */
void *buf;  /* Pointer to buffer */
```

ctlptr points to the structure that describes the control part (if any) to be included in the message: **buf** points to the buffer wherein the control information resides, and **len** gives the number of bytes to be sent.

Likewise, *dataptr* specifies the data (if any) to be included in the message. *flags* gives the message's type; it is described in detail below.

To send the data part of a message, *dataptr* must not be NULL, and the value of *dataptr*.**len** must be no less than zero. To send the control part of a message, the corresponding values must be set for *ctlptr*. **putmsg()** does not send the data portion of the message if *dataptr* is set to NULL or *dataptr*.**len** equals -1; likewise, **putmsg()** does not send the control portion of the message if *ctlptr* is NULL or *ctrlptr*.**len** equals -1.

If a control part is specified and *flags* equals **RS_HIPRI**, **putmsg()** sends a high-priority message. If no control part is specified and *flags* equals **RS_HIPRI**, **putmsg()** fails and sets **errno** to **EINVAL**. If *flags* is set to zero, **putmsg()** sends a message of normal priority. If neither the control part nor the data part is specified, and if flags is set to zero, **putmsg()** sends no message and returns zero.

The stream head guarantees that the control part of a message generated by **putmsg()** is at least 64 bytes long.

putmsg() usually blocks if the stream head's write queue is full due to internal flow-control conditions. For high-priority messages, **putmsg()** does not block on this condition. For other messages, **putmsg()** does not block when the write queue is full and you have set the mode on *fd* to **O_NDELAY** or **O_NONBLOCK**. **putmsg()** never sends a partial message. For details on **O_NDELAY** and **O_NONBLOCK**, see the Lexicon entry for **open()**.

Upon successful completion, **putmsg()** returns zero. If something goes wrong, **putmsg()** returns -1 and sets **errno** to an appropriate value. **putmsg()** fails if any of the following conditions is true:

- A non-priority message was specified, the mode on *fd* was set to **O_NDELAY** or **O_NONBLOCK**, and the stream-write queue is full due to internal flow-control conditions. **putmsg()** sets **errno** to **EAGAIN**.
- *fd* is not a valid file descriptor. **putmsg()** sets **errno** to **EBADF**.
- *ctlptr* or *dataptr* contains an illegal address. **putmsg()** sets **errno** to **EFAULT**.
- Your application caught a signal while it was executing **putmsg()**. **putmsg()** sets **errno** to **EINTR**.
- *flags* contains an undefined value, or you set *flags* **RS_HIPRI** but did not supply a control part. **putmsg()** sets **errno** to **EINVAL**.
- The stream referenced by *fd* is linked below a multiplexor. **putmsg()** sets **errno** to **EINVAL**.
- **putmsg()** could not allocate buffers for the message it was to send due to insufficient STREAMS memory resources. **putmsg()** sets **errno** to **ENOSR**.
- *fd* does not identify a stream. **putmsg()** sets **errno** to **ENOSTR**.
- A hangup condition was generated downstream for the specified stream, or the other end of the pipe is closed. **putmsg()** sets **errno** to **ENXIO**.
- The size of the message's data portion does not fall within range of legal packet sizes set by topmost stream module, or its control portion exceeds the maximum configured size. **putmsg()** sets **errno** to **ERANGE**.

putmsg() also fails if a STREAMS error message had been processed by the stream head before the call to **putmsg()** was executed. **putmsg()** returns the the value contained in the STREAMS error message.

See Also

getmsg(), libc, STREAMS

putp() — terminfo Function

Write a string into the standard window
#include <curses.h>
putp(*string***)**
char **string**;

COHERENT comes with a set of functions that help you read **terminfo** descriptions to manipulate a terminal. **putp()** writes the *string* into the standard window. It is equivalent to **tputs(***string***, 1, putchar);**.

See Also

curses.h, terminfo, tputs()

puts() — STDIO Function (libc)

Write string onto standard output
#include <stdio.h>
int puts(*string***)**
char **string*

puts() appends a newline character onto the string to which *string* points, and writes the result onto the standard output. If all goes well, it returns a nonnegative value (not necessarily -1); if an error occurs, it returns EOF.

Example

The following uses **puts()** to write a string on the screen.

```
#include <stdio.h>

main()
{
      puts("This is a string.");
}
```

See Also

fputs(), libc
ANSI Standard, §7.9.7.10
POSIX Standard, §8.1

Notes

For historical reasons, **fputs()** outputs the string unchanged, whereas **puts()** appends a newline character.

pututline() — General Function (libc)

Write a record into a logging file
#include <utmp.h>
struct utmp **pututline(record)*
const struct utmp **record;*

Function **pututline()** writes *record* into the file that logs login events. It is designed to update a record within the logging file.

record points to the record to be insert into the logging file. It is of type **utmp**, which is a structure whose fields describe a login event. (For a detailed description of this structure, see the Lexicon entry for **utmp.h**.)

pututline() assumes that you have first called **getutent()**, **getutid()**, or **getutline()** to open the logging file, and that the file's seek pointer is is at or before the record you wish to update. **pututline()** looks for the first record within the logging file whose field **ut_line** matches *record***.ut_line**. If it finds such a record, **pututline()** overwrites it with the contents of *record*; otherwise, it appends *record* onto the end of the logging file.

If all goes well, **pututline()** returns the address *record*. It returns NULL if the logging file had not been opened, or if it could not write *record* into the logging file.

By default, **getutid()** updates record in the logging file **/etc/utmp**. If you wish to manipulate another file, use the function **utmpname()**.

See Also

libc, utmp.h

putw() — STDIO Function (libc)

Write word into stream
#include <stdio.h>
int putw(*word***,** *fp***)**
int *word***; FILE** **fp;*

putw() writes *word* into the file stream to which *fp* points.

putw() differs from the related routine **putc()** in that **putw()** writes an **int**, whereas **putc()** writes a **char** that is promoted to an **int**.

By default, **putw()** returns the value written. If an error occurs, it returns EOF. You may need to call **ferror()** to distinguish this value from a genuine end-of-file flag.

See Also

ferror(), libc

Notes

Because **putw()** is implemented as a macro as well as a function, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

pwd — Command

Print the name of the current directory
pwd

pwd prints the name of the directory that you are in.

See Also

cd, commands, ksh, sh

Notes

Under the Korn shell, **pwd** is an alias for the expression **print -r $PWD**.

pwd.h — Header File

Define password structure
#include <pwd.h>

The header file **pwd.h** defines the structure **passwd**, which is used to build COHERENT's password file. **passwd** is defined as follows:

```
struct passwd {
        char    *pw_name;       /* login user name */
        char    *pw_passwd;     /* login password */
        int     pw_uid;         /* login user id */
        int     pw_gid;         /* login group id */
        int     pw_quota;       /* file quota (unused) */
        char    *pw_comment;    /* comments (unused) */
        char    *pw_gecos;      /* (unused) */
        char    *pw_dir;        /* working directory */
        char    *pw_shell;      /* initial program */
};
```

For detailed descriptions of the above fields, see the entry for **passwd**.

See Also

endpwent(), getpwent(), getpwnam(), getpwuid(), header files, setpwent()
POSIX Standard, §9.2.2