i-node — Definition

COHERENT system file identifier

Each file on a COHERENT file system is identified by a unique number, called an *i-node number* or *i-number*. Each i-node contains information about a file: its mode, link count, user identifier, group identifier, size, location on the file system, access time, modify time, and creation time.

The user refers to a file by a file name, stored in a directory; the directory entry identifies the file by its i-node number. A device and i-node number together uniquely specify a file. The headers **ino.h** and **i-node.h** define, respectively, disk i-nodes and memory i-nodes.

See Also

Using COHERENT

icheck — Command

i-node consistency check
icheck [-s] [-b *N ...***] [-v]** *filesystem ...*

Each block in a file system must be either free (i.e., in the free list) or allocated (i.e., associated with exactly one i-node). **icheck** examines each specified *filesystem*, printing block numbers that are claimed by more than one i-node, or claimed by both an i-node and the free list. It also checks for blocks that appear more than once in the block list of an i-node or in the free list.

The option **-v** (verbose) causes **icheck** to print a summary of block usage in the *filesystem*. The option **-s** causes **icheck** to ignore the free list, to note which blocks are claimed by i-nodes, and to rebuild the free list with the remainder. A list of block numbers may be submitted with the **-b** flag; **icheck** prints the data structure associated with each block as the file system is scanned.

The raw device should be used, and the *filesystem* should be unmounted if possible. If this is not possible (e.g., on the root file system) and the **-s** option is used, the system must be rebooted immediately to expunge the obsolete superblock.

The exit status bits for a bad return are as follows:

0x01 Miscellaneous error (e.g. out of space)
0x02 Too hard to fix without human intervention
0x04 Bad free block
0x08 Missing blocks
0x10 Duplicates in free list
0x20 Bad block in free list

See Also

clri, commands, dcheck, fsck, ncheck, sync, umount

Diagnostics

The message “dups in free” indicates a block is in the free list more than once. “bad freelist” indicates the presence of bad blocks on the free list. A “bad” block is one that lies outside the bounds of the file system. A “dup” (duplicated) block is one associated with the free list and an i-node, or with more than one i-node. All the errors above *must* be corrected before the file system is mounted. “bad ifree” means allocated i-nodes are on the free i-node list; this is inconsequential.

This command has largely been replaced by **fsck**.

id — Command

Print user and group IDs and names
id

The command **id** prints the user's real user ID and group ID. It also prints the effective IDs if they differ from the real IDs.

See Also

commands, getuid(), geteuid(), getgid(), getegid()

idbld — Command

Reconfigure the COHERENT kernel
/etc/conf/bin/idbld [-o *kernelname* **]**

The command **/etc/conf/bin/idbld** lets you reconfigure the entire COHERENT kernel. It systematically invokes all **mkdev** scripts in the subdirectories of **/etc/conf**. Each **mkdev** script, in turn, walks you through the task of formatting of one COHERENT's device drivers. This duplicates much of the work you performed when you first installed COHERENT onto your system.

After all of the **mkdev** scripts have been run, **idbld** invokes command **/etc/conf/bin/idmkcoh** to create a new kernel. Option **-o** tells **idbld** to name the new kernel *kernelname*. If you do not name this option, **idbld** by default names the new kernel **/coherent**.

See Also

commands, idenable, idmkcoh, idtune

ideinfo — Command

Display information of an IDE hard-disk drive
ideinfo [-c] /dev/at??

The command **ideinfo** displays information about device **/dev/at**??, which names a partition on an IDE hard disk. For example, command

```
ideinfo /dev/at0a
```

displays information about the first IDE drive on your system (drive 0). Among other things, this command displays the disk's manufacturer, the number of cylinders, header, sectors, and the number of bytes per sector on the disk.

Option **-c** tells **ideinfo** also to display how the device is partitioned.

See Also

at [device driver], commands

Notes

This command fails if the device is not an IDE hard drive.

idenable — Command

Enable or disable a device driver
/etc/conf/bin/idenable [-f *file***] [-de]** *driver*

The command **idenable** lets you enable or disable a device driver within the COHERENT kernel. *driver* is the device driver to enable or disable

The flag **-e** tells **idenable** to enable *driver*. This is the default.

The flag **-d** tells **idenable** to disable it.

For example, to enable STREAMS and disable the pseudo-tty driver **pty**, use the following commands:

```
/etc/conf/bin/idenable streams
/etc/conf/bin/idenable -d pty
```

idenable's command line can name more than one driver. For example, the command

```
/etc/conf/bin/idenable streams -d pty
```

is the equivalent of the two commands given above. The command line is parsed from left to right, so whatever you say last about a driver is what ultimately happens.

The option **-f** forces **idenable** to enable a driver. If **idenable** is directed to enable a device that will conflict with another enabled device in some way, it normally reports the conflict and not make the change. **-f** directs **idtune** to "force" the driver to be enabled by simply shutting off all other drivers with which a conflict occurs. For example, this is used with keyboard drivers, only one of which can occupy a major number at a time.

To implement your changes, you must then invoke the command **/etc/conf/bin/idmkcoh** to build a new kernel, which will reflect your changes, and then boot the new kernel.

idenable works by modifying the file **/etc/conf/sdevice**. It consists of a series of lines with the following format:

```
streams N   0   0   0   0   0x0   0x0   0x0   0x0
console Y   0   0   0   0   0x0   0x0   0x0   0x0
cohmain Y   0   0   0   0   0x0   0x0   0x0   0x0
```

The first column names the driver in question. The second column indicates whether it is incorporated into the kernel. The other columns give "magic cookies" that describe how the driver works.

You can read **/etc/conf/sdevice** to see how your kernel is currently configured. Note, however, that you must *never* modify **sdevice** by hand. **idenable** performs consistency checking to ensure, for example, that you do not load two competing keyboard drivers or hard-disk drivers. If you modify **sdevice** by hand, you run the risk of building a kernel that that will not boot or will trash your file system.

See Also

cohtune, commands, device drivers, idmkcoh, idtune, vtkb, vtnkb

idle — Device

Device that returns system's idle time
/dev/idle

/dev/idle is the device from which you can read the system's idle time. It has major device 0, the same as **/dev/null** and **/dev/cmos**; and has minor number 11. This non-portable device node is used exclusively for tracking system load. Its driver recognizes the system calls **open()**, **ioctl()**, and **close()**, but not **read()** or **write()**.

The only available **ioctl()** for **/dev/idle** writes a pair of **long**s to an address that you supply. The **long** at the lower address contains the number of system idle clock ticks (or, more precisely, the number of ticks at the end of which the system was idle) that have occurred since system startup. The **long** at the higher address contains the total number of clock ticks that have occurred since system startup. To estimate system load during a specific interval of time, perform the **ioctl()** for **/dev/idle** at the start and end of an interval.

Example

The following program prints system load over a five-second interval. To see a nonzero load percentage, run it concurrently with a CPU-intensive process.

```
#include <sys/null.h>

main()
{
      long x[2];  /* tick values at start of interval */
      long y[2];  /* tick values at end of interval */

      long delta_idle, delta_lbolt;
      int fd;

      /* We need to open a device before we can ioctl it. */
      fd = open("/dev/idle", 0);

      /* Get tick values at start of interval. */
      ioctl(fd, NLIDLE, x);

      /* Sleep during the interval. */
      sleep(5);
```

```
        /* Get tick values at end of interval. */
        ioctl(fd, NLIDLE, y);

        /* Compute number of system idle ticks during the interval. */
        delta_idle = y[0] - x[0];

        /* Compute total number of clock ticks during the interval. */
        delta_lbolt = y[1] - x[1];

        /* System is loaded when it isn't idle, so system load factor
         * is 100% minus the percentage of system idle time.
         */
        printf("system load = %ld%%\n",
              100L - (100L * delta_idle)/delta_lbolt);
        close(fd);
}
```

See Also

device drivers, ioctl(), null

idmkcoh — Command

Build a new kernel
idmkcoh [-o *kernelfile* **]**

The command **idmkcoh** creates a new bootable kernel. The kernel incorporates any changes that you may have made with the commands **idenable**, **idtune**, or **cohtune**. For details on how to use these commands, see their entries in the Lexicon. The changes you have made will take effect as soon as you boot the kernel that **idmkcoh** creates.

By default, **idmkcoh** writes the new kernel into file **/coherent**. The option **-o** tells **idmkcoh** to write the kernel into file **kernelfile** instead.

See Also

cohtune, commands, idbld, idenable, idtune, mdevice, mtune, sdevice, stune

idtune — Command

Set a tunable system value
/etc/conf/bin/idtune [-fm] *switch value*

The command **idtune** lets you "tune" a variable in the COHERENT kernel. It also performs some sanity checking, to help ensure that you do not set a value to an impossible value. It and the related command **cohtune** largely replace the need for the command **patch**.

To use **idtune**, simply invoke it along with the variable you wish to modify and the value to which you wish to set it. For example, to change the maximum size of a shared-memory segment to 128,000 bytes, type the command:

```
/etc/conf/bin/idtune SHMMAX 128000
```

For the new setting to come into effect, you must use the command **/etc/conf/bin/idmkcoh** to build a new kernel, and then boot the newly built kernel.

idenable recognizes the following two command-line options:

-f **idtune** by default will ask you if you are sure that you want to make a given change. This option suppresses that behavior.

-m Check that the value of *switch* is no less than *value*. If the value *switch* is less than *value*, then **idtune** raises it to *value*; otherwise, it leaves the value of *switch* alone.

idtune works by modifying the file **/etc/conf/stune**, which holds the values of system variables that users can set. **stune** consists of a series of entries like the following:

```
LOOP_COUNT          16
DUMP_USERS          2
MONO_COUNT          0
VGA_COUNT           4
```

The allowed range of values for a given variable is set in file **/etc/conf/mtune**, which consists of a series like the following:

```
STREAMS_HEAP    8192    32768   131072
MONO_COUNT      0       4       8
VGA_COUNT       0       4       8
NBUF_SPEC       0       0       5000
NHASH_SPEC      0       1021    5000
NINODE_SPEC     0       128     1024
NCLIST_SPEC     0       64      1024
```

The first column gives the variable, the second gives its minimum allowable value, the third gives its default value, and the last its maximum value.

You can read **mtune** and **stune** to see what kernel variables you can set, and to find the range of values allowed for each. Note, however, that you must *never* modify **stune** or **mtune** by hand. If you do so, you may build a kernel that is unbootable or that trashes your file system.

See Also

cohtune, commands, idenable, idmkcoh

ieee_d() — General Function (libc)

Convert a double from DECVAX to IEEE format
int
ieee_d(*idp***,** *ddp***)**
double **idp*, **ddp*;*

ieee_d() converts a **double** from DECVAX format to IEEE format. *ddp* points to a DECVAX-format **double** to convert. *idp* points to a destination for the converted IEEE value. *idp* may be identical to *ddp* for in-place conversion. The DECVAX significand is truncated, not rounded.

ieee_d() always returns zero, because the conversion always succeeds.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_d(), decvax_f(), float, ieee_f(), libc

ieee_f() — General Function (libc)

Convert a float from DECVAX to IEEE format
int
ieee_f(*ifp***,** *dfp***)**
float **ifp*, **dfp*;*

ieee_f() converts a **float** from DECVAX format to IEEE format. *dfp* points to a DECVAX-format **float** to convert. *ifp* points to a destination for the converted IEEE value. *ifp* may be identical to *dfp* for in-place conversion. The DECVAX significand is truncated, not rounded.

ieee_f() always returns zero, because the conversion always succeeds.

For a description of the IEEE and DECVAX formats for floating-point numbers, see the Lexicon article for **float**.

See Also

decvax_d(), decvax_f(), float, ieee_d(), libc

if — Command

Execute a command conditionally
if *sequence1* **then** *sequence2* **[elif** *sequence3* **then** *sequence4***] ... [else** *sequence5***] fi**

The shell construct **if** executes commands conditionally, depending on the exit status of the execution of other commands.

First, **if** executes the commands in *sequence1*. If the exit status is zero, it executes the commands in *sequence2* and terminates. Otherwise, it executes the optional *sequence3* if given, and executes *sequence4* if the exit status is zero. It executes additional **elif** clauses similarly. If the exit status of each tested command sequence is nonzero, it executes the optional **else** part *sequence5*.

Because the shell recognizes a reserved word only as the unquoted first word of a command, each **then**, **elif**, **else**, and **fi** must either occur unquoted at the start of a line or be preceded by ';'.

The shell executes **if** directly.

Example

For an example of this command, see the entry for **trap**.

See Also

commands, ksh, sh, test

if — C Keyword

Introduce a conditional statement

if is a C keyword that introduces a conditional statement. For example,

```
if (i==10)
      dosomething();
```

will **dosomething** only if **i** equals ten.

if statements can be used with the statements **else if** and **else** to create a chain of conditional statements. Such a chain can include any number of **else if** statements, but only one **else** statement.

See Also

C keywords, else
ANSI Standard, §6.6.4.1

IFS — Environmental Variable

Characters recognized as white space

The environmental variable **IFS** lists the characters that the shell recognizes as white space.

See Also

environmental variables, ksh, sh

index() — String Function (libc)

Find a character in a string
#include <string.h>
char *index(*string***,** *c***) char ****string***; char** *c***;**

index() scans the given *string* for the first occurrence of the character *c*. If *c* is found, **index()** returns a pointer to it. If it is not found, **index()** returns NULL.

Note that having **index()** search for a NUL character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

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

See Also

libc, pnmatch(), strchr(), string.h, strrchr(), string.h

Notes

You *must* include header file **string.h** in any program that uses **index()**, or that program will not link correctly.

index() is now obsolete. You should use **strchr()** instead.

inet_addr() — Sockets Function (libsocket)

Transform an IP address from text to binary
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
ulong inet_addr(*ip_address***)**
char ******ip_address***;***

The function **inet_addr()** translates an Internet-protocol (IP) address from text into binary format. *ip_address* gives the address where the string that holds the IP address resides in memory.

If all goes well, **inet_addr()** returns the binary address that it built from *ip_address*. If, however, *ip_address* points to a malformed Internet address, **inet_addr()** returns -1.

An IP address consists of four bytes. The four bytes normally are written as four numbers that are separated by periods; for example, "199.3.32.100". This way of rendering an IP address is called *dot notation*. Each byte can as a written as a decimal, octal, or hexadecimal number. By default, a numbers is written in decimal; a leading "0x" or "0X" indicates hexadecimal, and a leading '0' indicates octal.

When **inet_addr()** translates an IP address from text into binary, it simply transforms the four numbers as written into four bytes, which it writes into the four bytes of an unsigned long (32-bit) integer, from left to right, without regard to the machine's byte ordering. This means, among other things, that you cannot perform arithmetic on the address that **inet_addr()** returns — not even to increment or decrement it.

The IP address to which *ip_address* points can have any of the following four forms:

first.second.third.fourth
first.second.third
first.second
first

When the string to which *ip_address* points specifies all four parts of the Internet address, **inet_addr()** writes all four, from left to right, into the long integer that it returns.

When *ip_address* points to a three-part address, **inet_addr()** interprets the last (third) part as a 16-bit value, which it writes into into the rightmost two bytes of the network address. When *ip_address* points to a two-part address, **inet_addr()** interprets the second part as a 24-bit, which it writes into the rightmost three bytes of the network address.

When *ip_address* points to a one-part address, **inet_addr()** simply transforms it into an integer without shuffling any bytes.

See Also

inet_network(), libsocket

Notes

Because COHERENT does not yet support networking, **inet_addr()** is a dummy function that always returns zero.

inet_network() — Sockets Function (libsocket)

Transform an IP address from text to an integer
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
ulong inet_network(*ip_address***)**
char ******ip_address***;***

Function **inet_network()** translates an Internet-protocol (IP) address from text into a long integer. *ip_address* gives the address where the string that holds the IP address resides in memory.

If all goes well, **inet_network()** returns the integer that it built from *ip_address*. If, however, *ip_address* points to a malformed Internet address, **inet_network()** returns -1.

An IP address consists of four bytes. The four bytes normally are written as four numbers that are separated by periods; for example, "199.3.32.100". This way of rendering an IP address is called *dot notation*. Each byte can as a written as a decimal, octal, or hexadecimal number. By default, a numbers is written in decimal; a leading "0x" or

"0X" indicates hexadecimal, and a leading '0' indicates octal.

Unlike the function **inet_addr()**, **inet_network()** translates *ip_addr* into an unsigned long (32-bit) integer. This is the form suitable for a network address.

See Also

inet_addr(), libsocket

inetd.conf — System Administration

Configure the Internet daemons

File **/etc/inetd.conf** holds information that configures the Internet daemons on your system.

See Also

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

infocmp — Command

De-compile a terminfo file
infocmp [*file ...* **]**

infocmp reads a set of compiled terminal information, decodes its contents, and writes the decoded information to the standard output. It does its best to recreate the **terminfo** source from which the set of information had been compiled.

file must hold compiled **terminfo** information. If no *file* is named on the command line, **infocmp** reads the standard input.

infocmp first seeks *file* in the directory named by the environmental variable **TERMINFO**. If this variable has not been set, it seeks *file* in the default directory **/usr/lib/terminfo**. Thus, you can type the command

```
infocmp ansipc
```

in any directory and **infocmp** builds the appropriate path on its own.

In case of emergency, the output of **infocmp** can be piped to the **terminfo** compiler **tic**.

See Also

commands, term, tic, terminfo

Notes

infocmp was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk, with additional changes by Mark Williams Company.

init — System Administration

System initialization
/etc/init

COHERENT invokes processes in special order. The kernel invokes the command **init** as the initial process in the system. **init** runs as long as the system remains up. **init** is the first process that the kernel starts. The kernel always gives this process identifier 1.

init has two primary tasks: First, it guides the system through the latter stages of booting and entering multi-user mode. Second, it launches the appropriate processes so that users can log in and log out of COHERENT correctly. The rest of this article describes how **init** performs these tasks.

Booting and Entering Multi-user Mode

The following that **init** performs as it guides the system through entering multi-user mode.

First, if file **/usr/adm/wtmp** exists, **init** records there the date and time at which the system is being booted.

init then executes the shell script **/etc/brc**. This script usually loads the keyboard table and invokes the command **fsck** to check the file systems for errors If this script returns zero, then **init** enters multi-user mode; if not, it spawns the single-user shell.

When the user at the console terminates the single-user shell (usually by typing **<ctrl-D>**), **init** executes script **/etc/rc** and brings the system up to the multiuser state. **/etc/rc** performs such chores as setting the time zone, removing stale temporary files and lock files, and initializing the modem. If you wish, it can invoke the command

accton to enable process accounting.

init then reads file **/etc/ttys**. For every local, enabled line, **init** spawns the command **getty** with two arguments: the name of the port, and its speed (as given in **/etc/ttys**). Before it spawns a **getty**, **init** sets the group number for a new process group.

For a remote line, **init** spawns another copy of itself, which waits for carrier detect. Each **init** process spawned for a remote line also spawns **getty** when it detects a carrier signal on its port. (Note that this use of a second **init** process is unique to COHERENT.)

init then waits for the termination of its child processes. If one of the **getty** processes terminates, **init** respawns it. If another process terminates, **init** waits to receive its return value, so the process does not become a "zombie".

Logging In Users

The following describes how **init** logs users in.

As mentioned in the previous section, **init** invokes process **getty** for each enabled device on the system. **getty** and passes it as arguments the speed and the device upon which it should run. **getty** waits until someone tries to log in. Under COHERENT, **getty** sets the tty's line speed and local-edit characters and prompts the user to log in. It then locks the port, and invokes **login** with what the user has typed.

At this point, the command **login** takes over the task of logging in the user. **login** first asks the user for his password. It then reads the encrypted password from file **/etc/passwd**. If the password consists of one asterisk '*', **login** then reads the encrypted password from file **/etc/shadow**. It then compares the retrieved password with what the user has typed.

If the user has entered his password correctly, **login** executes various "housekeeping" tasks needed to get the user up and running under COHERENT. These include It records in file **/usr/adm/utmp** the fact of the user's logging in, which lets the system keep a running talley of who is logged into the system. For details on how **login** manages the task of logging in, see its entry in the Lexicon.

As its last action, **login** invokes the program named in **/etc/passwd**. This usually is an interactive shell (i.e., **sh** or **ksh**), but can also be another program (e.g., **uucico**). If **login** invokes an interactive shell, it does so with the first character of its **argv[0]** set to '-', so that the shell knows that it is a login shell. (For example, if **login** invokes **ksh**, its **argv[0]** is **-ksh**.)

The shell first executes file **/etc/profile**, then **$HOME/.profile**. Once these are executed, the shell displays its command-line prompt, and the user is ready to begin issuing commands to COHERENT

When the login shell terminates, **init** removes its record from file **/usr/adm/utmp**. Then it reopens the appropriate terminal and invokes **getty**, as described above. The device is now ready to receive another login.

Signals

init accepts two signals. When it receives **SIGQUIT**, it re-reads **/etc/ttys**, spawns **getty**s on newly enabled devices, and stops **getty**s on disabled devices. The command

```
kill quit 1
```

sends **SIGQUIT** to the **init** process. When **init** receives **SIGHUP**, it sends **SIGKILL** to every process and brings the system down to single-user mode. The command

```
kill -1 1
```

sends **SIGHUP** to the **init** process.

Files

/dev/console — Console terminal
/dev/tty?? — Terminal devices
/etc/rc — initialization command file
/etc/brc — Boot command file
/etc/ttys — Active terminals
/etc/utmp — Logged in users
/usr/adm/wtmp — Login accounting data
/usr/spool/uucp/LCK..* — Terminal locks

LEXICON

See Also

Administering COHERENT, getty, kill, ksh, login, sh, ttys

initgroups() — General Function (libc)

Initialize the supplementary group-access list
#include <sys/types.h>
#include <grp.h>
int initgroups(*user*, *basegid***)**
const char **user*; **gid_t** *basegid*;

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. The function **initgroups()** initializes the supplemental group-access list to the groups to which *user* belongs.

user is the login identifier of the user in question. *basegid* identifies that user's base group, as set in the file **/etc/passwd**. **initgroups()** calls the library function **getgrent()** to read from **/etc/group** all of the groups to which *user* belongs (in addition to her base group). It then calls **setgroups()** to initialize the supplementary group-access list to *user*'s base group and the additional groups returned by **getgrent()**.

If all goes well, **initgroups()** modifies the supplementary group-access list returns zero. Otherwise, it does not modify the list, returns -1, and sets **errno** to an appropriate value.

See Also

getgrent(), libc, setgroups()

Notes

If *user* belongs to more than **NGROUPS_MAX** groups, **initgroups()** reads only the first **NGROUPS_MAX** groups from **/etc/group** and ignores all of the others. Note that **NGROUPS_MAX** is a limit set by the POSIX Standard. For a fuller discussion of these limits, see the Lexicon entries for **sysconf()** and **limits.h**.

Only the superuser **root** can use **initgroups()**.

initialization — Definition

The term *initialization* refers to setting a variable to its first, or initial, value.

Rules of Initialization

Initializers follow the same rules for type and conversion as do assignment statements.

If a static object with a scalar type is not explicitly initialized, it is initialized to zero by default. Likewise, if a static pointer is not explicitly initialized, it is initialized to NULL by default. If an object with automatic storage duration is not explicitly initialized, its contents are indeterminate.

Initializers on static objects must be constant expressions; greater flexibility is allowed for initializers of automatic variables. These latter initializers can be arbitrary expressions, not just constant expressions. For example,

```
double dsin = sin(30.0);
```

is a valid initializer, where **dsin** is declared inside a function.

To initialize an object, use the assignment operator '='. The following sections describe how to initialize different classes of objects.

Scalars

To initialize a scalar object, assign it the value of a expression. The expression may be enclosed within braces; doing so does not affect the value of the assignment. For example, the expressions

```
int example = 7+12;
```

and

```
int example = { 7+12 };
```

are equivalent.

Unions and Structures

The initialization of a **union** by definition fills only its *first* member.

To initialize a **union**, use an expression that is enclosed within braces:

```
union example_u {
      int member1;
      long member2;
      float member3;
} = { 5 };
```

This initializes **member1** to five. That is to say, the **union** is filled with an **int**-sized object whose value is five.

To initialize a structure, use a list of constants or expressions that are enclosed within braces. For example:

```
struct example_s {
      int member1;
      long member2;
      union example_u member3;
};

struct example_s test1 = { 5, 3, 15 };
```

This initializes **member1** to five, initializes **member2** to three, and initializes the *first* member of **member3** to 15.

Strings

To initialize a string pointer,
use a string literal.

The following initializes a string:

```
char string[] = "This is a string";
```

The length of the character array is 17 characters: one for every character in the given string literal plus one for the null character that marks the end of the string.

If you wish, you can fix the length of a character array. In this case, the null character is appended to the end of the string only if there is room in the array. For example, the following

```
char string[16] = "This is a string";
```

writes the text into the array **string**, but does not include the concluding null character because there is not enough room for it.

A pointer to **char** can also be initialized when the pointer is declared. For example:

```
char *strptr = "This is a string";
```

initializes **strptr** to point to the first character in **This is a string**. This declaration automatically allocates exactly enough storage to hold the given string literal, plus the terminating null character.

Arrays

To initialize an array, use a list of expressions that is enclosed within braces. For example, the expression

```
int array[] = { 1, 2, 3 };
```

initializes **array**. Because **array** does not have a declared number of elements, the initialization fixes its number of elements at three. The elements of the array are initialized in the order in which the elements of the initialization list appear. For example, **array[0]** is initialized to one, **array[1]** to two, and **array[2]** to three.

If an array has a fixed length and the initialization list does not contain enough initializers to initialize every element, then the remaining elements are initialized in the default manner: static variables are initialized to zero, and other variables to whatever happens to be in memory. For example, the following:

```
int array[3] = { 1, 2 };
```

initializes **array[0]** to one, **array[1]** to two, and **array[2]** to zero.

The initialization of a multi-dimensional array is something of a science in itself. The ANSI Standard defines that the ranks in an array are filled from right to left. For example, consider the array:

```
int example[2][3][4];
```

This array contains two groups of three elements, each of which consists of four elements. Initialization of this array will proceed from **example[0][0][0]** through **example[0][0][3]**; then from **example[0][1][0]** through **example[0][1][3]**; and so on, until the array is filled.

It is easy to check initialization when there is one initializer for each "slot" in the array; e.g.,

```
int example[2][3] = {
      1,  2,  3,  4,  5,  6
};
```

or:

```
int example[2][3] = {
     {  1,  2,  3 }, {  4,  5,  6 }
};
```

The situation becomes more difficult when an array is only partially initialized; e.g.,

```
int example[2][3] = {
       { 1 }, { 2, 3 }
};
```

which is equivalent to:

```
int example[2][3] = {
     { 1, 0, 0 }, { 2, 3, 0 }
};
```

As can be seen, braces mark the end of initialization for a "cluster" of elements within an array. For example, the following:

```
int example[2][3][4] = {
     5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
     { 2, 3, 7 } };
```

is equivalent to entering:

```
int example[2][3][4] = {
     { 5, 0, 0, 0 },
     { 1, 2, 0, 0 },
     { 5, 2, 4, 3 },

     { 9, 9, 5, 0 },
     { 2, 3, 7, 0 },
     { 0, 0, 0, 0 }
};
```

The braces end the initialization of one cluster of elements; the next cluster is then initialized. Any elements within a cluster that have not yet been initialized when the brace is read are initialized in the default manner.

See Also

array, C language, Programming COHERENT, struct, union
ANSI Standard, §3.5.7

ino.h — Header File

Constants and structures for disk i-nodes
#include <sys/inode.h>

inode.h declares structures and constants that are used to describe i-nodes.

See Also

i-node, header files

inode.h — Header File

Constants and structures for memory-resident i-nodes
#include <sys/inode.h>

inode.h declares structures and constants for memory-resident i-nodes.

See Also

header files, i-node

install — Command

Install a software update onto COHERENT
/etc/install [-c] *id device ndisks*

The command **install** installs an update of the COHERENT system onto your hard disk. *id* identifies the update to be installed. *device* is the device from which the update disks will be read. *ndisks* is the number of disks that comprise the update.

Option **-c** tells **install** to uncompress compressed files directly from the installation disks, rather than copy the compressed files onto disk and uncompress them there. **install** reads information about compression formats and options from file **/etc/install.u**. This switch permits software packages other than COHERENT to use compressed files.

Third-Party Software

install also provides a standard mechanism by which software developers can install their software onto systems that run COHERENT. The rest of this article discusses how to prepare a software release so that it can be installed using **install**.

For **install** to be able to install a software distribution, the distribution must consist of a set of mountable floppy disks, each holding a COHERENT file system created by **mkfs**. This keeps the disks independent of each other and also lets the user to insert the disks in any order. **install** records the fact that it has read a given disk from the distribution, thus preventing the user from attempting to read a given disk more than once during an installation session.

Floppy disks should be built using **mkfs**. Each disk in the distribution must hold in its root directory a file whose name is of the form:

/id.sequence

Here, *id* identifies the release, as described above. Note that *id* must be formed from the set of upper- and lower-case letters, digits, the period '.', and the underscore character '_', and not exceed nine characters in length. *sequence* indicates which disk in the distribution this disk is, from one through the total number of disks.

install uses the command **cpdir** to copy each of the distribution disks to directory **/** on the current system. Therefore, every disk should be "root based" (i.e., full path names should be used). Because **install** is run by the superuser, **cpdir** preserves the date and time for each file, and preserves ownership and modes. To keep file ownership consistent with COHERENT conventions, make files that are neither *setuid* nor *setgid* owned by user **bin** and group **bin**. **install** recreates on your hard disk all directories that it finds on the distribution disks, as needed. Be careful when choosing the ownership and mode of directories because you could inadvertently compromise the security of your users' systems.

Preprocessing

When you load a disk, **install** seeks a file named *id.disk*.**pre** upon that disk. If it finds such a file, **install** assumes that that file is a script, copies it into directory **/conf**, and executes it from there *before* it copies any files from the disk onto your system. If you are installing COHERENT, it uses the command:

id.disk.**pre** *id.disk*.**arch**

If you are updating a package rather than installing it, **install** uses the command:

id.disk.**pre -u** *id.disk*.**arch**

install always gives the same argument to the **.pre** script. As its suffix **.arch** indicates, the argument usually names a file whose contents name files that should be archived **install** copies the contents of the disk onto your system. **install** merely passes the name of the **.arch** file that *might* exist on the installation disk to the **.pre** script: it is up to the **.pre** script to check for the existence of the **.arch** file, read it, and perform the archiving. Of course, the **.pre** script can ignore this argument should it choose.

For example, if you are installing X Windows onto your COHERENT system, the identifier is **CohX**. When you load the first disk into your system, **install** looks for file **CohX.1.pre** on that disk. If it finds that file, **install** copies it into directory **/conf** on your root file system, and invokes it with the command:

```
/conf/CohX.1.pre CohX.arch
```

All of this occurs *before* **install** copies any files from that disk onto your system. In this way, files can be archived or otherwise backed up before they are overwritten by the package you are installing onto your system.

One last behavior should be noted: if **install** finds a **.pre** file on the *first* disk of the installation kit, it also seeks on that disk a file that has the suffix **.supp** on that disk. The suffix **.supp** stands for "suppression": normally, it names files that are *not* to be copied from the release onto your system. It is the flip side of the **.arch** file.

Note that you can mount the disks of a release and edit these **.arch** and **.supp** files *before* you install the package onto your system. In this way, you can protect your system from being damaged by installing new software onto your system.

Postprocessing

After all disks in a distribution have been successfully copied by the user, **install** checks for the existence of a file of the form

```
/conf/id.post
```

where *id* matches the *id* field found on the **install** command line. If found, **install** executes this file to allow special "postprocessing," such as installing manual pages into directory **/usr/man** or executing installation-specific commands.

Before an installation procedure completes its postprocessing, it should remove any *id* files of the following form from the target system:

/conf/*id*.post
/*id.sequence*

Adding Manual-Page Entries

As part of building a distribution, you usually must generate pre-processed or "cooked" manual-page entries for distribution with your upgrade or add-on package. These should be inserted into the subdirectories of **/usr/man**, with the name of the subdirectory being specific to your product. This naming convention avoids name-space collisions, should multiple applications use the same name for a manual-page entry.

If you install new or additional manual pages, you must update the index file used by the **man** command to locate manual entries. File **/usr/man/man.index** on the target file system contains index entries for all manual pages on the system. As part of postprocessing, you must append index information for your manual pages to the end of the existing index file. In addition, file **/usr/man/man.help** contains the **man** command's help message. This includes a list of valid topics and some explanatory text. You should also append to this file a brief list of the manual page entries that you have added. For further information on manual pages, see the Lexicon entry for the command **man**.

Logging

install logs all partial as well as completed installations in file **/etc/install.log**. This information includes date/time stamps and the command-line arguments to **install**.

Example

The following installs COHERENT update **coh.301**, which consists of one disk, from a high-density 5.25-inch floppy drive:

```
/etc/install coh.301 /dev/fha0 1
```

Files

/etc/install.log

See Also

commands, man, mkfs

int — C Keyword

Data type

An **int** is the most commonly used numeric data type, and is normally used to encode integers. Under COHERENT 386, an **int** is the same size as a **long**; **sizeof int** equals 4 (31 bits plus a sign bit), and can hold any value from -2,147,483,647 to 2,147,483,647. Under COHERENT 286, an **int** is the same size as a **short**; that is, **sizeof int**

equals 2 (15 bits plus a sign bit), and can hold any value from from -32768 to +32767.

An **int** normally is sign extended when cast to a larger data type; an **unsigned int**, however, will be zero extended.

See Also

C keywords, data formats, data types, long, short
ANSI Standard, §6.1.2.5

interrupt — Definition

An **interrupt** is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

See Also

Programming COHERENT, signal()

io.h — Header File

Constants and structures used by I/O
#include <sys/io.h>

io.h declares constants and structures used by various I/O routines.

See Also

header files

ioctl() — System Call (libc)

Device-dependent control
#include <unistd.h>
#include <*header***.h>**
ioctl(*fd***,** *command***,** *arg***)**
int *fd***,** *command***; char** **arg***;*

ioctl() lets you interact directly with a device driver. You can use it to set or retrieve parameters for devices (line printers, communications lines, terminals), and non-standard spacing operations for tape drives.

ioctl() acts upon the block-special file or character-special file associated with the file descriptor *fd*. *command* points to the specific request.

header names the header file that defines symbolic commands for the device you wish to manipulate. Using the symbolic command definitions from the header files promotes device independence within each device type. A complete list of symbolic commands appears below.

arg passes a buffer of information (defined by structures in the appropriate header files) to the driver. For any *command* not needing additional information, this argument should be NULL.

Some **ioctl()** requests work on all files, and are not passed to any driver.

ioctl() returns -1 on errors, such as a bad file descriptor. Because the call is device dependent, almost any other error could be returned.

Commands

The following gives the commands that can be used with **ioctl()**, as extracted from COHERENT's header files. Please note the following caveats:

- New drivers are being added continually to COHERENT, both by Mark Williams Company and by users and third-party vendors. You should regard the following list as being tentative at best.
- Because the commands and arguments with with **ioctl()** are unique to COHERENT's suite of device drivers, **ioctl()** is one of the least portable of all system calls. If you want your code to run on multiple operating systems, you should use **ioctl()** judiciously.

<sys/cdrom.h>
Header file used to manipulate a CD-ROM device. Unless otherwise noted, *arg* is ignored:

CDROMPAUSE Pause playing an audio CD.

CDROMRESUME	Resume playing an audio CD.
CDROMPLAYMSF	Play an audio CD at a given minute-second frame (MSF) address. *arg* points to an array of six bytes that give the MSF address.
CDROMPLAYTRKIND	Play a track on an audio CD. *arg* points to an array of four bytes that give, respectively, the start track, the start index, the end track, and the end index of the track to be played.
CDROMREADTOCHDR	Read the CD's table-of-contents header. *arg* points to a structure of type **cdrom_tochdr** into which the header is written.
CDROMREADTOCENTRY	Read an entry from the table-of-contents header. *arg* points to a structure of type **cdrom_tocentry** into which the entry is written.
CDROMSTOP	Spin down the CD-ROM drive's motor.
CDROMSTART	Turn on the CD-ROM drive's motor.
CDROMEJECT	Eject the CD-ROM. Note that this does not work on every variety of CD-ROM drive.
CDROMVOLCTRL	Control the volume on an audio CD. *arg* points to an array of four bytes that, respectively, set the the volume on channels zero through three.
CDROMSUBCHNL	Read data about a sub-channel. *arg* points to a structure of type **cdrom_subchnl** into which the information about the sub-channel is written.
CDROMREADMODE1	Read type-1 data. *arg* points to a structure into which the data are written.
CDROMREADMODE2	Read type-2 data. *arg* points to a structure into which the data are written.

<sys/fdioctl.h>

This header file is used with the floppy-disk drive:

FDFORMAT	Format a track on a floppy disk. *arg* points to a two-byte array that identifies, respectively, the cylinder and head to format.

<sys/hdioctl.h>

This header file is used with AT-style hard-disk drives (i.e., IDE, ESDI, MFM, or RLL disks). *arg* gives the address in user memory where drive attributes reside, or to which they should be written:

HDGETA	Get drive attributes.
HDSETA	Set drive attributes.
HDGETIDEINFO	Get the attributes of an IDE drive. *arg* should point to a copy of the structure **ide_info**; this call to **ioctl()** initializes the structure with the requested information.

<sys/null.h>

This header file defines **ioctl**s that examine system memory:

NLFREE	Read the amounts of memory on your system that are available and free. *arg* gives the address of an object of type **FREEMEM**, which is defined in header file **<null.h>**. This type is an array of two **long**s: the first receives the amount of available memory, and the second the amount of free memory. For an example of a program that uses this **ioctl()**, see the Lexicon entry for **freemem**.
NLIDLE	Read the system's idle time. *arg* points to an array of two **long**s. The first **long** receives system's idle ticks; the second, the number of ticks since system startup. From reading these values repeatedly, you can compute the changes in system idle time and time since startup, and so see what the system's load is. For an example of how to this call to **ioctl()**, see the Lexicon entry for **idle**.

<sys/sdioctl.h>

The commands defined in this header file are passed to the driver **aha**, which manipulates Adaptec SCSI disks. None does anything.

<sgtty.h>

The following commands are used with the **sgtty** method of controlling terminal devices. They are documented in more detail in the Lexicon entry for **sgtty**. *arg* points to a structure of type **sgttyb**, which is defined in that header file:

TIOCHPCL	Hang up on last close.
TIOCGETP	Get modes (old **gtty**).
TIOCSETP	Set modes (old **stty**).
TIOCSETN	Set modes without delay or flush.

TIOCEXCL	Set exclusive use.
TIOCNXCL	Set non-exclusive use.
TIOCFLUSH	Flush I/O queues.
TIOCSETC	Set characters.
TIOCGETC	Get characters.

<stropts.h>

STREAMS commands. *arg* points to a STREAMS control block that will be used to generate an **M_IOCTL** message.

I_NREAD	Get message length, count.
I_PUSH	Push named module.
I_POP	Pop topmost module.
I_LOOK	Get name of the topmost module.
I_FLUSH	Flush read/write side.
I_SRDOPT	Set stream head read mode.
I_GRDOPT	Get stream head read mode.
I_STR	Send **ioctl()** message downstream.
I_SETSIG	Register for signal **SIGPOLL**.
I_GETSIG	Return registered event mask.
I_FIND	Locate named module on stream.
I_LINK	Link two streams.
I_UNLINK	Unlink two streams.
I_RECVFD	Receive file descriptor from pipe.
I_PEEK	Examine stream head data.
I_SENDFD	Send file descriptor to pipe.

The following commands are not covered by iBCS2:

I_SWROPT	Set stream write mode.
I_GWROPT	Get stream write mode.
I_LIST	Get name of all modules/drivers.
I_PLINK	Create persistent link.
I_PUNLINK	Undo persistent link.
I_FLUSHBAND	Flush priority band.
I_CKBAND	Check for existence of priority band.
I_GETBAND	Get band of first message.
I_ATMARK	Check whether current message is marked.
I_SETCLTIME	Set drain timeout for stream.
I_GETCLTIME	Get the current close timeout.
I_CANPUT	Check if band is writeable.

<sys/tape.h>

Header file for interfacing with magnetic-tape devices. *arg* points to an area in user space that holds additional information for the tape device. A tape driver may recognize any of the following **ioctl()** commands:

T_ERASE	Erase tape.
T_LOAD	Load. Not used.
T_RDSTAT	Read status.
T_RST	Reset.
T_RETENSION	Retension tape.
T_RWD	Rewind tape.
T_SBB	Space block backward — move backward by *arg* blocks. Not used.
T_SBF	Space Block Forward — move forward by *arg* blocks. Not used.
T_SBREC	Not used.
T_SFB	Space Filemark Backward — move backwards by *arg* files.
T_SFF	Space Filemark Forward — move forward by *arg* files.
T_SFREC	Not used.
T_TINIT	Not used.
T_UNLOAD	Unload. Not used.
T_WRFILEM	Write file marks. Not used.

<termio.h>

The following commands are used with the **termio** method of controlling a terminal. They are documented in more detail in the Lexicon entry for **termio**. *arg* points to a structure of type **sgttyb**, which is described above.

TCGETA	Get terminal parameters.
TCSETA	Set terminal parameters.
TCSETAW	Wait for drain, set parameters.
TCSETAF	Wait for drain, flush input, set parms.
TCSBRK	Send 0.25-second break.

The following commands also take arguments when called via **ioctl()**:

TCXONC	Start/stop control: An argument of zero suspends output; an argument of one restarts suspended output.
TCFLSH	Flush queues: An argument of zero flushes the input queue; an argument of one flushes the output queue; and an argument of two flushes both queues.

<sys/vtkd.h>

This header file defines commands used with the keyboard driver. *arg* points to a structure of type **sgttyb**, which is defined in header file **sgtty.h**.

KDMAPDISP	Map the display into user space.
KDSKBMODE	Toggle the scan code **xlate**.
KDMEMDISP	Dump a byte of virtual or physical memory.
KDGKBSTATE	Get the keyboard's shift state.
KIOCINFO	Determine the workstation of the virtual terminal.
KIOCSOUND	Start sound generation.
KDGETLED	Get the state of the keyboard's LEDs.
KDSETLED	Set the state of the LEDs.

The following four **ioctl()** commands allow user programs to perform I/O instructions directly, rather than going through the system-call interface and having the kernel perform the I/O. The most common need for these functions is in window managers and similar applications, where the usual kernel interface would be unacceptably slow.

Normally, any user program that attempts to execute I/O instructions directly to hardware will get an immediate **SIGSEGV** and be terminated. Use of the commands below allow user-level programs to perform I/O without being terminated. The I/O operations are available through functions **inb()**, **outb()**, etc., which are present in the kernel-support library **/etc/conf/lib/k386.a** and are documented in the manual to the COHERENT Device Driver Kit.

Access to any of these functions may be restricted to the superuser on some systems:

KDENABIO	Allow the user process permission to perform input/output operations to all available I/O addresses. The third argument to **ioctl()** is ignored.
KDDISABIO	Prohibit user processes from performing input/output operations to all available I/O addresses. The third argument to **ioctl()** is ignored. It is normal for direct I/O to ports to be disallowed at user level. The main reason for this call is to undo the effect of preceding **KDENABIO** or **KDADDIO** calls.
KDADDIO	Allow user-level I/O to a port. The third argument to **ioctl()** is an **unsigned short** that gives the single address value of the port.
KDDELIO	Disallow user-level I/O to a port. The third argument to **ioctl()** is an **unsigned short** that gives the single address value of the port. It is normal for direct I/O to ports to be disallowed at user level. The main reason for this call is to undo the effect of preceding **KDADDIO** calls.

Example

The following program, by Udo Munk, demonstrates how to use **ioctl()** to read a mouse plugged into a serial port. It takes one argument, the name of the port you wish to check.

```
#include <fcntl.h>
#include <poll.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <termio.h>

char *mouse;
int mouse_fd;
struct termio old_tty, new_tty;

/* do the right thing by signals */
sig_handler()
{
      ioctl(mouse_fd, TCSETAF, &old_tty);
      exit(EXIT_SUCCESS);
}

/* cry and die */
void fatal(message)
char *message;
{
      fprintf (stderr, "%s\n", message);
      exit(EXIT_FAILURE);
}

/* run the whole shebang */
main(argc, argv)
int argc; char **argv;
{
      struct pollfd fds[1];

      if (argc != 2)
            fatal ("Usage: findmouse /dev/com[1-4]pl");

      if (strncmp(argv[1], "/dev/com1pl", 11) &&
                  strncmp(argv[1], "/dev/com2pl", 11) &&
                  strncmp(argv[1], "/dev/com3pl", 11) &&
                  strncmp(argv[1], "/dev/com4pl", 11))
            fatal ("Usage: findmouse /dev/com[1-4]pl");

      mouse = argv[1];

      signal(SIGINT, sig_handler);
      signal(SIGQUIT, sig_handler);
      signal(SIGHUP, sig_handler);

      fprintf(stdout, "Trying to open %s ...\n", mouse);
      if ((mouse_fd = open(mouse, O_RDONLY)) < 0)
            fatal ("Cannot open this device.");
      fprintf(stdout, "Success.\n");

      fprintf(stdout, "Trying to read line mode of %s ...\n", mouse);
      if (ioctl(mouse_fd, TCGETA, &old_tty) < 0)
            fatal ("Cannot read this device’ss line mode.");
      fprintf(stdout, "Success.\n");

      new_tty = old_tty;
      new_tty.c_cflag &= ~(CBAUD | HUPCL);
      new_tty.c_cflag |= CLOCAL | B1200;
      new_tty.c_iflag = IGNBRK;
      new_tty.c_oflag = new_tty.c_lflag = 0;

      /*
       * VMIN = 0, VTIME = 0 has the same effect as setting O_NDELAY on the
       * input line.
       */
      new_tty.c_cc[VMIN] = 0;
      new_tty.c_cc[VTIME] = 0;
```

```
      /* Set up to poll the input line. */
      fds->fd = mouse_fd;
      fds->events = POLLIN;

      fprintf(stdout, "Trying to set new line mode for %s ...\n", mouse);
      if (ioctl(mouse_fd, TCSETAF, &new_tty) < 0)
            fatal ("Cannot set new tty line mode");
      fprintf(stdout, "Success.\n");

      fprintf(stdout, "\nI'm reading from %s. To exit, type <ctrl-C>.\n",
            mouse);
      fprintf(stdout,
            "If you see stuff on the screen when you move the mouse,\n");
      fprintf(stdout,
            "then you have found the mouse port.\n");
      fprintf(stdout, "\nNow wiggle your mouse:\n");

      for (;;) {
            size_t read_count;
            unsigned char mousebuf [128];

            /* Block waiting for mouse input. */
            if (poll (fds, 1, -1) < 0)
                  break;

            /* Drain input in large chunks until it becomes time to block. */
            while ((read_count = read (mouse_fd, mousebuf,
                        sizeof (mousebuf))) != 0) {
                  unsigned char * scan = mousebuf;

                  do
                        printf ("%02x ", * scan ++);
                  while (-- read_count != 0);

                  fflush (stdout);
            }
      }
}
```

See Also

device drivers, exec, getty, header files, libc, open(), read(), sgtty, stty(), termio

Notes

The type of the *arg* to **ioctl()** is declared as **char** * mainly to improve portability. In most cases, the actual argument type will be something like **struct sgttyb** *, depending on the device and command. The actual argument should be cast to type **char** * to ensure cross-machine portability.

Under COHERENT 286, the main header file for **ioctl()** is **<sgtty.h>**. This header file is also included with COHERENT 386 for compatibility with older applications.

ipc.h — Header File

Definitions for interprocess communications
#include <sys/ipc.h>

ipc.h defines constants and structures used by functions that perform interprocess communications.

See Also

header files, msgget(), semget(), shmget()

ipcrm — Command

Remove an interprocess-communication memory item
ipcrm [-mqs *id***] [-MQS** *key***]**

The command **ipcrm** removes a memory item used for interprocess communication: either shared-memory segment, message queue, or semaphore set. You can use this command either with an *id*, which is the identifying number assigned by the function that created the memory item in question; or with a *key*, which is the identifier used by the application that requested the memory item.

ipcrm recognizes the following options:

m *id* Remove the shared-memory segment with an identifier of *id*.

q *id* Remove the message queue with an identifier of *id*.

s *id* Remove the semaphore set with an identifier of *id*.

M *key* Remove the shared-memory segment with a key of *key*.

Q *key* Remove the message queue with a key of *key*.

S *key* Remove the semaphore set with a key of *key*.

To find the identifiers and keys for for the IPC resources that are currently allocated, use the command **ipcs**.

See Also

commands, ipcs, msgget(), semget(), shmget()

Notes

ipcrm does not remove a shared-memory segment until all processes attached to it are removed by calls to the function **shmat()**.

Any user can run **ipcrm**; however, a user can remove only those memory items that he "owns", as noted in the control structure for the item. The superuser **root** can, of course, remove all memory items, no questions asked.

ipcs — Command

Display a snapshot of interprocess communications
ipcs [-abcmopst] [-N *kernel***]**

The command **ipcs** prints information about interprocess communication (IPC) objects.

Options

ipcs recognizes the following command-line options:

-a "All" print option; i.e., combine the options **-b**, **-c**, **-o**, **-p**, and **-t**.

-b "Biggest" option: Display the maximum size that the kernel allows for each kind of IPC object.

-c Display the login name and group name of the user who created each IPC object.

-m Display information about shared-memory segments. By default, **ipcs** displays information about all IPC objects.

-N *kernel*
Read kernel-specific information from *kernel* instead of from the default kernel **/autoboot**.

-o "Outstanding usage" option: Display the number of messages currently queued and their total size in bytes, and the number of processes attached to each shared-memory segment.

-p Display the process identifiers of the following:

- The last process to send a message.
- The last process to receive a message on each message queue.
- Each creating process.
- The last process to attach to or detach from each shared-memory segment.

-q Display information about message queues. By default, **ipcs** displays information about all IPC objects.

-s Display information about sets of semaphores. By default, **ipcs** displays information about all IPC objects.

-t Display the following information about times:

- When functions **msgsnd()** and **msgrcv()** were last executed for each message queue.
- When the functions **shmat()** and **shmdt()** were last executed for each shared-memory segment.
- When the function **semop()** was last executed for each set of semaphores.

Format of Displayed Information

The following names and describes each column of information that **ipcs** can display for each IPC object. The letters in parentheses name the command-line options tell **ipcs** to display this column; **all** means that **ipcs** always displays this column:

ATIME (**-at**)
The last time a process attached itself to this shared-memory segment.

CBYTES (**-ao**)
The total number of bytes in this message queue.

CGROUP (**-ac**)
The name of the group to which the creator of this IPC object belongs.

CPID (**-ap**)
The identifier of the process that created this shared-memory segment.

CREATOR (**-ac**)
The login identifier of the user who created this IPC object.

CTIME (**-at**)
The time when this IPC object was created or last changed.

DTIME (**-at**)
The most recent time a process detached itself from this shared-memory segment.

GROUP (**all**)
The name of the group to which the owner of this IPC object belongs.

ID (**all**) The numeric identifier of this IPC object.

KEY (**all**)
The key that names this IPC object. Applications use this key to identify and manipulate the IPC object.

LPID (**-ap**)
The identifier of the last process to have attached itself to or detached itself from this shared-memory segment.

LRPID (**-ap**)
The identifier of the last process to have received a message from this message queue.

LSPID (**-ap**)
The identifier of the last process to have sent a message to this message queue.

MODE (**all**)
The IPC object's mode. The mode is a string of 11 characters that interprets the value of field **mode** in the structure **ipc_perm**, which is part of each IPC object. (For more information on this structure, see the Lexicon entries **msgget()**, **semget()**, and **shmget()**.) The first two mode characters are as follows:

R A process is waiting for **msgrcv()**.
S A process is waiting for **msgsnd()**.
D The associated shared-memory segment has been removed.
C The associated shared-memory segment will be cleared when the first process attaches itself to it.
- The corresponding flag is not set.

The last nine characters of the mode give the permissions on the IPC object — three sets of three characters each. In each set, the first character marks whether read permission is granted, the second whether permission to write or alter is granted, and the third is unused. The first set gives the permissions of the user who created the object (its "owner"); the second, the permissions of other users in the owner's group; and the third, the permissions of all other users.

NATTCH (**-ao**)
The number of processes attached to this shared-memory segment.

NSEMS (**-ab**)
The number of semaphores in this set.

OTIME (..... t)
The last time a semaphore operation was completed on this set.

OWNER (**all**)
The login identifier of the user who "owns" this IPC object.

QBYTES (**-ab**)
The number of bytes left available to the messages in this queue.

QNUM (**-ao**)
The number of messages in this queue.

RTIME (**-at**)
The last time a message was received from this queue.

SEGSZ (**-ab**)
The size of this shared-memory segment.

STIME (**-at**)
The last time a message was sent to this queue.

T (**all**) The type of IPC object this is, as follows:

m	Shared-memory segment
q	Message queue
s	Set of semaphores

See Also

commands, ipcrm, msgget(), semget(), shmget()

Notes

ipcs gives information about the way interprocess communications are at the moment you run it. The data it returns can change even as you read them.

IRQ — Technical Information

Interrupts on the IBM PC

The term *IRQ* stands for "interrupt request". The IBM PC has 16 interrupts channels built into it. Some are reserved for system hardware; most are available for cards and peripheral devices. The following gives the default assignments for IRQs:

IRQ	*Device*
0	System timer
1	Keyboard controller
2	Second IRQ controller
3	Serial port (COM) 2
4	COM1
5	Line printer (LPT) 2 or LPT3
6	Floppy-disk controller
7	LPT1
8	Real-time clock
9	Re-directed IRQ2
10	Available
11	Available
12	Motherboard mouse port (available if no mouse)
13	Mathematics coprocessor
14	Hard-disk (AT) controller
15	Available

As you can see, there are two banks of interrupt controllers, each of which controls eight interrupts, with IRQ2 latched to the first port on the second chip, IRQ9.

Channel 5 handles two parallel ports — LPT2 and LPT3. If you install three serial ports onto your system, be careful on how you jumper the card, or you will confuse your system.

Due to the design of the PC, IRQ 7 can display spurious interrupts when a device signals an IRQ line other than 7, then cancels the signal before the interrupt controller figures out which line the IRQ occurred on. Thus, you

should not assign other devices to IRQ 7, if at all possible.

Three interrupt channels are available for user hardware: channels 10, 11, and 15. Channel 12 is also available if you do not have a bus mouse.

Only two interrupts are available for serial ports, COM1 and COM2. Note that COM3 and COM4 are "linked" to COM1 and COM2, respectively. For this reason, if you have both COM1 and COM3 your system, or both COM2 and COM4, only one of the pair can be interrupt driven; the other port of the pair must be polled.

See Also

Administering COHERENT

isalnum() — ctype Function (libc)

Check if a character is a number or letter
#include <ctype.h>
int isalnum(*c***) int** *c***;**

isalnum() tests whether the argument *c* is alphanumeric (**0-9**, **A-Z**, or **a-z**). It returns a number other than zero if *c* is of the desired type, and zero if it is not. **isalnum()** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.1
POSIX Standard, §8.1

isalpha() — ctype Function (libc)

Check if a character is a letter
#include <ctype.h>
int isalpha(*c***) int** *c***;**

isalpha() tests whether the argument *c* is a letter (**A-Z** or **a-z**). It returns a number other than zero if *c* is an alphabetic character, and zero if it is not. **isalpha()** assumes that *c* is an ASCII character or EOF.

Example

For an example of this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.2
POSIX Standard, §8.1

isascii() — ctype Function (libc)

Check if a character is an ASCII character
#include <ctype.h>
int isascii(*c***) int** *c***;**

isascii() tests whether *c* is an ASCII character (0 <= *c* <= 0177). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many **ctype** macros fail if passed a non-ASCII value other than EOF.

Example

For an example of how to use this function, see the entry for **ctype.h**.

See Also

ASCII, libc

Notes

Please note that **isascii()** is not part of the ANSI standard. Programs that use it may not be portable to all implementations of C.

isatty() — General Function (libc) (libc)

Check if a device is a terminal
#include <unistd.h>
int isatty(*fd***) int** *fd***;**

isatty() checks to see if a device is a terminal. It returns one if the file descriptor *fd* describes a terminal, and zero otherwise.

Files

/dev/* — Terminal special files
/etc/ttys — Login terminals

See Also

ioctl(), libc, tty, ttyname(), ttyslot(), unistd.h
POSIX Standard, §4.7.2

iscntrl() — ctype Function (libc)

Check if a character is a control character
#include <ctype.h>
int iscntrl(*c***) int** *c***;**

iscntrl() tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns a number other than zero if *c* is a control character, and zero if it is not. **iscntrl()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

libc
ANSI Standard, §7.3.1.3
POSIX Standard, §8.1

isdigit() — ctype Function (libc)

Check if a character is a numeral
#include <ctype.h>
int isdigit(*c***) int** *c***;**

isdigit() tests whether the argument *c* is a numeral (**0-9**). It returns a number other than zero if *c* is a numeral, and zero if it is not. **isdigit()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.4
POSIX Standard, §8.1

isgraph() — ctype Function (libc)

Check if a character is printable
#include <ctype.h>
int isgraph(int *c***);**

isgraph() tests whether *c* is a printable letter within the ASCII character set, but excluding the space character. The ANSI Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isgraph() returns nonzero if *c* is a printable character (except for space), and zero if it is not.

See Also

ASCII, libc
ANSI Standard, 7.3.1.5
POSIX Standard, §8.1

islower() — ctype Function (libc)

Check if a character is a lower-case letter
#include <ctype.h>
int islower(*c***) int** *c***;**

islower() tests whether the argument *c* is a lower-case letter (**a-z**). It returns a number other than zero if *c* is is a lower-case letter, and zero if it is not. **islower()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.6
POSIX Standard, §8.1

ispos() — Multiple-Precision Mathematics (libmp)

Return if variable is positive or negative
#include <mprec.h>
int ispos(*a***)**
mint **a***;*

ispos() returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative.

See Also

libmp

isprint() — ctype Function (libc)

Check if a character is printable
#include <ctype.h>
int isprint(*c***) int** *c***;**

isprint() is a macro that tests if *c* is printable, i.e, if it is neither a delete nor a control character. It returns a number other than zero if *c* is a printable character, and zero if it is not. **isprint()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.7
POSIX Standard, §8.1

ispunct() — ctype Function (libc)

Check if a character is a punctuation mark
#include <ctype.h>
int ispunct(*c***) int** *c***;**

ispunct() tests whether the argument *c* is a punctuation mark, i.e., neither an alphanumeric character nor a control character. It returns a number other than zero if the character tested is a punctuation mark, and zero if it is not. **ispunct()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.8
POSIX Standard, §8.1

isspace() — ctype Function (libc)

Check if a character prints white space
#include <ctype.h>
int isspace(*c***) int** *c***;**

isspace() tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns a number other than zero if *c* is a white-space character, and zero if it is not. **isspace()** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.9
POSIX Standard, §8.1

isupper() — ctype Function (libc)

Check if a character is an upper-case letter
#include <ctype.h>
int isupper(*c***) int** *c***;**

isupper() tests whether the argument *c* is an upper-case letter (**A-Z**). It returns a number other than zero if *c* is an upper-case letter, and zero if it is not. **isupper()** assumes that *c* is an ASCII character or **EOF**.

Example

For an example of how to use this macro, see the entry for **ctype.h**.

See Also

ASCII, libc
ANSI Standard, §7.3.1.10
POSIX Standard, §8.1

isxdigit() — ctype Function (libc)

Check if a character is a hexadecimal numeral
#include <ctype.h>
int isxdigit(*c***)**
int *c***;**

isxdigit() tests whether *c* is a hexadecimal numeral — that is, any of the characters '0' through '9', any of the letters 'a' through 'd', or any of the letters 'A' through 'D'. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isxdigit() returns nonzero if *c* is a hexadecimal numeral, and zero if it is not.

See Also

ASCII, libc
ANSI Standard, §7.3.1.11
POSIX Standard, §8.1

itom() — Multiple-Precision Mathematics (libmp)

Create a multiple-precision integer

#include <mprec.h>
mint *itom(*n***)**
int *n***;**

itom() creates a new multiple-precision integer (or **mint**), initializes it to the signed integer value *n*, and returns a pointer to it. You can use the function **mintfr()** to reclaim the storage used by the **mint** created by **itom()**.

See Also

libmp