backups — Technical Information

Strategies for backing up COHERENT

This entry describes how to backup files — that is, how to copy one or more selected files onto floppy disks. You should do this regularly to provide yourself with a spare copy of valuable files should your system suffer a catastrophe.

The strategy you adopt for backups will vary quite a bit, depending upon the medium onto which you back up your files: tapes or floppy disks. Floppy disks are inexpensive, but their limited capacity means that you have to plan carefully. Tapes are simpler to use than floppy disks, but are more expensive. The following sections describe first the strategies for backing up onto floppy disks; and then for backing up onto tapes.

Backing up Onto Floppy Disks

There are two general strategies for backing up files onto floppy disks:

- Use the command **tar** to create archives of files on a floppy disk. This is fine for archiving a limited set of files on an irregular basis.
- The other strategy uses the command **gnucpio** to implement a system of regular dumps. This strategy is preferred for systems that daily amass data of importance for a real-world job, such as running a business or managing a research project.

You should always have a procedure of backups for your system. Which strategy you use depends on how you are using your system. The following sections describe how to implement each strategy of backups. Note that COHERENT includes a version of the UNIX utility **dump** for the sake of compatibility with older versions of UNIX and COHERENT; however, **dump** is obsolete, should not be used, and will not be described here.

Please note that the following descriptions assume that you are using a 5.25-inch, high-density floppy disks set in drive 0 (drive A). For a list of available floppy-disk devices, see the Lexicon entry for **floppy disks**.

The following describes how to use **tar** to back up onto floppy disks.

The first step is to prepare floppy disks to receive files. Insert a 5.25-inch floppy disk into drive 0, and then type the following command:

```
/etc/fdformat -v /dev/rfha0
```

The command **fdformat** formats the floppy disk, verifying that no media defects exist. You must perform this task of formatting a floppy disk before you use it the first time.

The next step is to create an archive of the files you wish to back up. Use the portable archive command **tar** to collect a mass of files into an archive on the floppy disks. For example, to archive all files in directory **source**, use the following command:

```
tar cvf /dev/rfha0 source
```

The options **cvf** tell **tar** to create an archive, run in verbose mode, and write the archive onto the device or into the file named in the next argument. **/dev/rfha0** names the floppy device onto which you wish to write the archive. Finally, **source** is the directory whose files you wish to back up.

To perform a listing of the contents of the newly created archive, type

```
tar tvf /dev/rfha0
```

The options **tvf** tell **tar** to list the contents of the archive, run in verbose mode, and read the archive from the device or file named in the next argument.

To extract several files from the archive, enter a command of the form

```
tar xvf /dev/rfha0 source/myfile 'source/*.c'
```

The options **xvf** tell **tar** to extract or unarchive the specified files, run in verbose mode, and read the archive from the device or file named in the next argument. Note that the second file argument contains a "wildcard" character and thus must be quoted to prevent expansion by the shell.

For more information on how to use **tar**, see its entry in the Lexicon.

The following describes how to back up using **gnucpio**.

The COHERENT utility **gnucpio** performs mass dumps and restores of files using a universally recognized file format.

In this example, dumps are performed monthly, weekly, and daily. You should prepare at least three sets of floppy disks for the monthly saves, giving you three months of full backup. You will use the floppy disks in rotation, with the oldest always used next.

Once a month, you should dump the entire system.

Once a week, you should dump information in the system that is new or has been changed since the end of the previous week. You will need five sets of floppy disks, because some months have five weekends in them.

Finally, every day you should save information that has changed that day. For these dumps, you will need five sets of floppy disks: one for each working day. You may need extras in case of weekend work.

Label each set of disks carefully as *monthly*, *weekly*, or *daily*. Label the daily floppy disks "Monday" through "Friday", the weekly floppy disks "Week 1" through "Week 5", and the monthly floppy disks "Month 1" through "Month 3". When you perform the dump, write the date on the label.

The following gives a step-by-step description of how to use **gnucpio** to back up files. The next samples are given with the suggestion that your system has only one 5.25-inch floppy-disk drive.

1. Log into the system as the superuser **root**.
2. If you have not yet done so, use the command **fdformat** to format a set of floppy disks, as shown above. With high-density, 5.25-inch floppy disks, a rule of thumb is to prepare one floppy disk for each megabyte of data to be dumped.
3. If other users are logged into the system, use the command **wall** to request that they log off. For example:

   ```
   /etc/wall
   Please log off.
   Time for file dump.
   <ctrl-D>
   ```

4. Be sure that all users are logged off the system by typing the command:

   ```
   who
   ```

   This command names all users who are still on the system.

   If they have not logged off in a few minutes, send another message. Repeat the process until **who** shows no users except yourself.

5. When all other users have logged off, execute the command **shutdown** as described in its Lexicon entry.
6. Run the script **mount.all** to mount all of your file systems. Then, run the COHERENT command **fsck** on each file system to check its integrity.
7. If this is the last workday of the month, perform a *monthly* dump, to back up the entire system. Insert the first volume of the correct monthly dump floppy disk into the floppy drive, after adding today's date to the label, and type the commands:

   ```
   cd /
   find . -print | gnucpio -ocF /dev/rfha0
   ```

   Option **-F** tells **gnucpio** to write everything to the raw, 2400-block, floppy-disk device **/dev/rfha0**.

Note that if you want to split your dump across different media (i.e., write the first volume onto tape and the second onto a floppy disk), you should not use the option **-F**; **gnucpio** will write its output to the standard output, and you can use the shell operator '>' to redirect that to the device **/dev/rfha0**. If you do not use **-F**, **gnucpio** will ask you, after it finishes writing a volume, for the name of the device into which it should redirect the next volume of output.

As more floppies are needed, **gnucpio** will ask you to insert them. Be sure to label each floppy disk with its volume number.

8. If this is the last work day of the week, but not the last workday of the month, perform a *weekly* dump. Prepare the correct weekly dump floppy disks, add today's date to the label, insert the first floppy disk, and type the command:

```
cd /
find . -newer cpio.weekly -print | gnucpio -ocF /dev/rfha0
touch cpio.weekly
```

This will dump all files that are younger than file **cpios.weekly**.

9. If this is neither the last workday of the month nor the last workday of the week, you will perform a *daily* dump. Prepare the daily dump floppy disk with today's day of the week, add today's date to the label, insert the first floppy disk into the drive, and type the command:

```
cd /
find . -newer cpio.daily -print | gnucpio -ocF /dev/rfha0
touch cpio.daily
```

This will dump files that are younger than file **cpio.daily**.

10. Type **sync** to ensure that all buffers are flushed.

11. When you are finished dumping data, type the command **/etc/reboot** to return your system to multi-user mode.

For more information on how to use **gnucpio** and **find**, see their respective entries in the Lexicon.

If you wish, you can back up only limited portions of your system. To do so, just name in your **find** command the directories you wish to back up. For example, to back up everything in your home directory and in **/usr/lib**, use the following command:

```
find $HOME /usr/lib -type f -newer cpio.daily -print | gnucpio -ocF /dev/rfha0
touch cpio.daily
```

When you determine the backup strategy you wish to use, you should save the appropriate commands into a script, to ensure that backups are run correctly every time.

The following describes how to restore files from floppy disks.

If you find that a file has been inadvertently destroyed, you can restore the information to disk from backup floppy disks.

To restore information from backups created with **gnucpio** or **tar**, you must first determine the date and time that the file was last known to have been modified. From this date, determine on which set of disks the file was last correctly dumped. Find the set of floppy disks labeled with that date, and insert into the floppy-disk drive the first one in the set. For example, if you wish to restore the file **myfile**, from a **gnucpio** archive, use the command:

```
gnucpio -icdvF /dev/rfha0 myfile
```

To retrieve **myfile** from a **tar** archive, use the command:

```
tar xvf /dev/rfha0 myfile
```

Both of these commands assume that the disks are high-density, 5.25-inch floppies in drive 0 (drive A). See the Lexicon article **floppy disk** for a table that shows which COHERENT device is associated with which size and density of disk, and which disk drive. You may have to insert more than one disk from the set of backups until you find the one that holds the file you want.

Backing up Onto Tapes

The strategy for backing up onto tape resembles that for floppy disks, with the exception that in many instances the tape medium is larger than the device being backed up. This makes it worth your while to back up the entire device every time you do a back up, rather than perform incremental backups. The reason for this is simple: the

fewer tapes over which you have spread your backups, the lower the risk that one will fail.

To back up an entire partition, do the following:

1. Pop a tape into your tape device. Make sure the tape is appropriately labeled.
2. Log in as the superuser **root**, and type the following command:

   ```
   /etc/shutdown single 0
   ```

   This returns your system to single-user mode immediately.
3. Use the command **gtar** to back up your partition, as follows:

   ```
   gtar -cvzf /dev/tape directory
   ```

   tape identifies the tape device onto which the backup will be written, and *directory* identifies the file system to back up. For example, tape device **/dev/rStp2** is a SCSI tape device that has SCSI identifier 2 and performs autorewinding. For a list of recognized tape devices, see the article for **tape**. in the Lexicon.

   Please note two points about *directory*. First, do *not* use the absolute path name when specifying a directory: that is, use **usr**, *not* **/usr**. **gtar** strips the leading '/' in any event, but it's always best to use relative path names whenever possible. Second, in single-user mode only the root file system is mounted by default; therefore, if the file system you wish to back up resides on its own partition, you must mount that file system by hand before you begin to back it up.

   Note that the **z** option to the **gtar** command tells **gtar** to use **gzip** to compress the files automatically. File compression is a good idea: because fewer bits are being written to the tape, the backup will go faster; and because less tape is used, the risk of a tape failure is lessened.
3. When **gtar** has finished writing to the tape, wait until the tape finishes rewinding; then remove it from its drive and put it in a safe place (i.e., away from magnets and children). Then type **<ctrl-D>** to return your system to multi-user mode.

That's all there is to it. To restore information from the tape, put the tape into the drive and use the **gtar** command to fetch the file you want. For example, to restore file **/v/fwb/myfile.c** from a SCSI tape drive that has SCSI identifier 2, use the following command:

```
gtar -xvzf /dev/rStp2 "v/fwb/myfile.c"
```

Note that the file will be written into a subdirectory of your current directory. For example, if your current directory is **/v/fwb**, then **myfile.c** will be restored into a file with the path name **/v/fwb/v/fwb/myfile.c**. This may be a little inconvenient, but is not nearly as inconvenient as having to create **myfile.c** by hand.

An Example of Using Floppy Tape

This section gives examples of how to use QIC-40/QIC-80 ("floppy tape") to write archives to floppy tape, and read them back. It uses the commands **tape**, which manipulates the tape device; and **gtar**, which writes archives onto the physical tape, and reads them back.

Suppose you have a directory named **dir1**, which contains files you want to backup. To back up all files in that directory onto a tape, insert a tape cartridge into the drive, then type:

```
gtar -cvf /dev/ft dir1
```

To verify that the contents of the tape match the original files, run **gtar** again in verification ("diff") mode:

```
gtar -df /dev/ft
```

We strongly urge you to verify tapes after they have been written, especially with floppy-tape devices. If a tape fails this test, throw it away and build a new archive; otherwise, you may receive a nasty surprise when you try to restore a file from that tape. Do not be surprised if an otherwise sound tape fails after time: a tape does wear out after a number of uses.

To later extract the files from the tape, use

```
gtar -xf /dev/ft
```

To use data compression, the preceding commands can be used with the addition of **gtar**'s option **-z**, as follows:

```
gtar -czvf /dev/ft dir1
gtar -dzf /dev/ft
gtar -xzf /dev/ft
```

To backup only selected files to tape, you could do the following:

```
find dir -type f -print | sort > Files
```

then manually edit the file **Files** so it contains only the names of the files you want to back up. Then use the command:

```
gtar -cv -T Files -f /dev/ft
```

The previous examples used **/dev/ft**, the device node that calls for the tape to be rewound when the device is closed. This is convenient if you are putting only one archive onto tape. To concatenate multiple archives on a single cartridge, use the no-rewind-on-close device. For example, suppose you have a second directory, **dir2**, and you want to back it up on the same tape, after an archive of **dir1**. The following commands accomplish this:

```
gtar -cvf /dev/nft dir1
gtar -cvf /dev/nft dir2
```

After each archive is written, the tape remains positioned at the end of the archive. To verify the contents of both archives, do the following:

```
# this command rewinds the tape:
tape rewind
# this command displays the contents of the first archive:
gtar -tvf /dev/nft
# this command displays the contents of the second archive:
gtar -tvf /dev/nft
```

If you make a note of the locations of archives as they are written, you can retrieve them later without having to read the preceding archives. For example:

```
# rewind the tape:
tape rewind
# write "dir1" archive at start of tape:
gtar -cvf /dev/nft dir1
# find current position of the tape:
tape tell
```

The command **tape tell** returns a string of the form:

```
Tape Is at Byte Offset 102400
```

Continuing:

```
# write "dir2" archive after "dir1":
gtar -cvf /dev/nft dir2
# read the current position:
tape tell
```

The second instance of **tape tell** returns a string of the form:

```
Tape Is at Byte Offset 235520
```

That is, it shows that the tape has advanced after the second archive was written onto it. At this point, the cartridge is removed, then reinserted into the tape drive at a later date:

```
tape seek 102400
gtar -tvf /dev/tape
```

The command **tape seek** moves the tape to the byte position **102400**, i.e., the end of the first archive. This command assumes that you jotted down the position displayed by the command **tape tell** executed earlier. The command **gtar** then displays the contents of the second archive.

See Also

Administering COHERENT, gnucpio, gtar, tape

bad — Command

Maintain list of bad blocks
bad [-acdl] *device* **[** *block ...* **]**

A hard disk or floppy disk may have bad blocks on it: a "bad block" is a portion of disk that is flawed, and so cannot reliably be read or written. It is the unusual disk that is free of bad blocks.

COHERENT keeps a list of bad blocks so it can avoid using them. The command **bad** maintains this bad-block list for the given *device*, which must be a block-special file. **bad** recognizes the following command-line options:

a Add each given *block* to the bad-block list
c Clear the bad-block list
d Delete each given *block* from the bad-block list
l List all blocks on the bad-block list

Note that **bad** merely adds a block to the list of bad blocks, or removes a block from that list. It does not deallocate any i-node associated with a block when adding it to the bad-block list. You should run the command **icheck** with the option **-s** immediately after **bad** to correct the problem, or run the command **fsck**. After you modify the list of bad blocks, you must reboot your system to force the kernel to use this modified list.

The file system on *device* should be unmounted if possible. You must have appropriate permissions for *device* before you can invoke **bad**. For many file systems, only the superuser may use **bad** to change the bad-block list. Use the command **badscan** to create a prototype file of bad blocks.

When the command **mkfs** creates a file system, the prototype specification may include a list of bad blocks for the new file system.

See Also

badscan, commands, icheck, mkfs

badscan — Command

Build bad block list
/etc/badscan [-v] [-o *proto* **] [-b** *boot* **]** *device size*
/etc/badscan [-v] [-o *proto* **] [-b** *boot* **]** *device xdevice*

badscan scans a floppy disk or a partition of the hard disk for bad blocks. It writes onto the standard output a prototype file that lists all bad blocks on the disk.

badscan recognizes the following options:

-v Print an estimate of time needed to finish examining the device.

-o *proto* Redirect output into file *proto*.

-b *boot* Insert a given *boot* into the proto file as the bootstrap. The default is **/conf/boot**.

device names the special device to scan.

The command line for **badscan** comes in two forms, as shown at the top of this article. The first version is for a floppy disk; *size* gives the size of the device, in blocks. The second version is for a hard-disk partition; *xdevice* specifies devices **/dev/at0x** or **/dev/at1x**, which hold the partition-table information for the disk in question. **badscan** reads the data from the boot block of the drive to find the size of the **device**.

Examples

The first example uses **badscan** to find all bad blocks on a high-density, 3.5-inch floppy disk in drive 1 (i.e., drive B), and writes its output into file **proto**:

```
/etc/badscan -v -o proto /dev/rfva1 2880
```

See the article **floppy disks** for a table that gives the device name and number of sectors to be found on the various types of floppy disk that COHERENT recognizes.

The second example uses **badscan** to prepare a list of bad blocks for partition 2 on hard-drive 0, which is an IDE drive accessed via COHERENT's **at** driver. Again, the output is written into file **proto**:

```
/etc/badscan -v -o /conf/proto.at0c /dev/rat0c /dev/at0x
```

See Also

at, bad, commands, floppy disks, mkfs

Notes

Because SCSI hard-disk drives maintain their own map of bad blocks, **badscan** is not required for SCSI drives. However, we recommend that you use it on removeable-media SCSI drives.

banner — Command

Print large letters
banner [*argument ...*]

banner prints large (seven-character by five-character) letters on the standard output. Each *argument* produces one large text output line. If there is no *argument*, each line from the standard input produces one line of large-text output.

See Also

commands, libmisc, lpr, pr

basename — Command

Strip path information from a file name
basename *file* [*suffix*]

basename strips its argument *file* of any leading directory prefixes. If the result contains the optional *suffix*, **basename** also strips it. **basename** prints the result on the standard output.

For example, the command

```
basename /usr/fred/source.c
```

returns

```
source.c
```

basename is most useful when it is used with other shell commands. For example, the command

```
for i in *.c
do
      cp $i `basename $i .c`.backup
done
```

copies every file that has the suffix **.c** into an identically named file that has the suffix **.backup**.

See Also

commands, ksh, sh

bc — Command

Interactive calculator with arbitrary precision
bc [-l] [*file ...*]

bc is a language that performs calculations on numbers with an arbitrary number of digits. **bc** is most commonly used as an interactive calculator, where the user types arithmetic expressions in a syntax reminiscent of C. If you invoke **bc** with no *file* argument, it reads the standard input. For example:

```
Input             Output
(1000+23)*42      42966
k = 2^10
16 * k            16384
2 ^ 100           1267650600228229401496703205376
```

You can invoke **bc** with one or more *file* arguments. After **bc** reads each *file*, it reads the standard input. This provides a convenient way to read programs that are stored in files. COHERENT includes a library of mathematical functions for **bc**; to use it, invoke **bc** with its option **-l**.

The following summarizes briefly the facilities provided by **bc**. More information is available in the tutorial to **bc** that is included with this manual.

The delimiters '**/***' and '***/**' enclose comments. Names of variables or functions consist of a lower-case letter followed by any number of letters or digits. (Names cannot begin with an upper-case letter because numbers with

a base greater than ten may need upper-case letters for their notation.) The three built-in variables **obase**, **ibase**, and **scale** represent, respectively, the number base for printing numbers (default, ten), the number base for reading numbers (default, ten), and the number of digits after the decimal (radix) point (default, zero). Variables may be simple variables or arrays, and need not be pre-declared, with the exception of variables internal to functions. Some examples of variables and array elements are **x25**, **array[10]**, and **number**.

Numbers are any string of digits, and may have one decimal point. Digits are taken from the ordinary digits (0-9) and then the upper-case letters (A-F), in that order.

Certain names are reserved for use as key words. The key words recognized by **bc** include the following:

if, for, do, while
Test conditions and define loops, with syntax identical to C

break, continue
Alter control flow within **for** and **while** loops.

quit Tell **bc** to exit immediately

define *function* **(***arg, ..., arg***)**
Define a **bc** function by a compound statement, as in C.

auto *var, ..., var*
Define variables that are local to a function, rather than having global scope.

return (*value***)**
Return a value from a function.

scale (*value***)**
Return the number of digits to the right of the decimal point in *value*.

sqrt (*value***)**
Return the square root of *value*

length (*value***)**
Return the number of decimal digits in *value*.

bc recognizes the following operators:

```
+       -       *       /       %       ^       ++
--      =       +=      -=      *=      /=      %=
^=      ==      !=      <       <=      >       >=
```

These operators are similar to those in C, with the exception of **^** and **^=**, which are exponentiation operators. Expressions can be grouped with parentheses. Statements are separated with semicolons or newlines, and may be grouped with braces into compound statements.

bc prints the value of any statement that is an expression but is not an assignment.

As in the editor **ed**, an '!' at the beginning of a line causes that line to be sent as a command to the COHERENT shell **sh**.

The library **lib.b** holds code written in **bc** for the following mathematical variables and functions:

atan(*z*) Arctangent of *z*
cos(*z*) Cosine of *z*
exp(*z*) Exponential function of *z*
j(*n,z*) *n*th order Bessel function of *z*
ln(*z*) Natural logarithm of *z*
pi Value of pi to 100 digits
sin(*z*) Sine of *z*

If you invoke **bc** with its option **-l**, it reads **lib.b** and thus makes the above functions and constants available to you.

Examples

The first example calculates the factorial of its positive integer argument by recursion.

```
/*
 * Factorial function implemented by recursion.
 */
define fact(n) {
      if (n <= 1) return (n);
      return (n * fact(n-1));
}
```

The second example also calculates the factorial of its positive integer argument, this time by iteration.

```
/*
 * Factorial function implemented by iteration.
 */
define fact(n) {
      auto result;

      result = 1;
      for (i=1; i<=n; i++) result *= i;
      return (result);
}
```

Files

/usr/lib/lib.b — Source code for the library

See Also

commands, conv, dc, libmp
bc Desk Calculator Language, tutorial

Notes

Line numbers do not accompany error messages in source files.

bc performs integer calculations with arbitrary precision, limited only by the memory available. However, the results of some calculations on numbers with fractional parts depends on the specified **scale**; see the tutorial for details.

bcmp() — String Function (libc)

Compare two chunks of memory
int bcmp (*source***,** *destination***,** *count***)**
VOID **source***, **destination***; **size_t** *count***;**

Function **bcmp()** compares the first *count* bytes of data at address *source* with the first *count* bytes of data at address *destination*. It returns the offset of the first character where *source* and *destination* differ; if they do not differ, it returns zero.

See Also

libsocket, memcmp()

Notes

This function is included for compatibility with Berkeley socket code. It is equivalent to the standard C function **memcmp()**, except that its first two arguments are reversed.

bcopy() — String Function (libc)

Berkeley function to copy memory
void bcopy(*source***,** *destination***,** *amount***)**
char **source***, **destination***;
int *size***;**

Function **bcopy()** copies *size* bytes of data from address *source* to address *destination*. *destination* must point to enough allocated memory to hold *size* bytes of data, or problems will result.

See Also

libc, memcpy(),

Notes

Please note the arguments of **bcopy()** are the opposite of those used by **memcpy()**. This function is included solely for compatibility with existing code; users are encouraged to use the standard function **memcpy()** instead.

bind() — Sockets Function (libsocket)

Bind a name to a socket
#include <sys/types.h>
#include <sys/socket.h>
int bind (*socket***,** *name***,** *namelen***)**
int *socket***,** *namelen***; struct sockaddr** **name***;*

Function **bind()** binds a name to an unnamed socket.

When function **socket()** creates a socket, that socket exists but has no name. **bind()** creates a special file, assigns it a name, and binds that file to a socket. Thereafter, the socket can be accessed by reading or writing the file.

socket is a file descriptor that identifies the socket in question. It must have been returned by a call to **socket()**. *name* points to the full path name of the file to which *socket* is to be bound. The calling process must unlink *name* when it no longer needs it. *namelen* gives the number of bytes in the path name name to which *name* points. Under COHERENT, no element of *name* can exceed 14 characters (not including separating '/' characters).

If all goes well, **bind()** returns zero. If something goes wrong, **bind()** returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **bind()** sets **errno**:

EBADF *socket* is somehow not a valid descriptor.

ENOTSOCK
socket is not a socket.

EADDRNOTAVAIL
name is not available from the local machine.

EADDRINUSE
name is already bound to another socket.

EINVAL
socket is already bound to a name.

EACCES
The memory to which *name* points is protected and the user lacks permission to access it.

EFAULT
name points to an illegal address.

ENOTDIR
The path name to which *name* points contains an element that is not a directory.

EINVAL
The path name to which *name* points contains a character with the high-order bit set.

ENOENT
A prefix component of the path name does not exist.

EIO An I/O error occurred while creating the directory entry for *name* or allocating its inode.

EROFS *name* would reside on a read-only file system.

EISDIR
name points to an empty path name.

Example

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

See Also

connect(), getsockname(), libsocket, listen(), socket()

bit — Definition

bit is an abbreviation for "binary digit". It is the basic unit of data processing. A bit can have a value of either zero or one. Bits can be concatenated to form bytes.

A bit can be used either as a placeholder to construct a number with an absolute value, or as a flag whose value has a particular meaning under specially defined circumstances. In the former use, a string of bits builds an integer. In the latter use, a string of bits forms a **map**, in which each bit has a meaning other than its numeric value.

See Also

bit map, byte, nybble, Programming COHERENT,
ANSI Standard, §1.6

bit-fields — Definition

A *bit-field* is a member of a structure or **union** that is defined to be a cluster of bits. It provides a way to represent data compactly. For example, in the following structure

```
struct example {
      int member1;
      long member2;
      unsigned int member3 :5;
}
```

member3 is declared to be a bit-field that consists of five bits. A colon ':' precedes the integral constant that indicates the *width*, or the number of bits in the bit-field. Also, the bit-field declarator must include a type, which must be one of **int**, **signed int**, or **unsigned int**.

A bit-field that is not given a name may not be accessed. Such an object is useful as "padding" within an object so that it conforms to a template designed elsewhere.

A bit-field that is unnamed and has a length of zero can be used to force adjacent bit-fields into separate objects. For example, in the following structure

```
struct example {
      int member1;
      int member2 :5;
      int :0;
      int member3 :5;
};
```

the zero-length bit-field forces **member2** and **member3** to be written into separate objects.

Finally, it is illegal to take the address of a bit-field.

See Also

bit, bit map, byte, Programming COHERENT,
ANSI Standard, §3.5.2.1

Notes

Because bit-fields have many implementation-specific properties, they are not considered to be highly portable. Bit-fields use minimal amounts of storage, but the amount of computation needed to manipulate and access them may negate this benefit. Bit-fields must be kept in integral-sized objects because many machines cannot directly access a quantity of storage smaller than a "word" (a word is generally used to store an **int**).

bit_count() — Sockets Function (libsocket)

Count bits in a bit-mask
int bit_count (*mask***)**
unsigned *mask***;**

The function **bit_count()** counts and returns the bits in *bitmask* that have been turned on.

See Also

libsocket

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value.

See Also

bit, byte, Programming COHERENT,

Notes

C permits the manipulation of bits within a byte through the use of bit-field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

block — Technical Information

A *block* is a mass of data that is read at one time. Blocks are different lengths under different operating systems; COHERENT defines a block as being **BSIZE** bytes long.

Information is read in blocks from block-special devices, such as the hard disk or floppy disks. This is done to increase the speed with which data are read from these devices; reading characters one at a time, such as is done with character-special devices such as terminals or modems, would be too slow.

See Also

Using COHERENT,
ANSI Standard, §3.6.2

boot — Driver

Boot block for hard-disk partition/nine-sector diskette

Several different programs are used to load COHERENT from a floppy or hard disk into memory. This process is called *bootstrapping* (from the old expression about pulling one's self up by one's bootstraps) or *booting* for short. The program used depends upon whether one is loading COHERENT from a hard-disk partition, from a 5.25-inch floppy disk, or from a 3.5-inch floppy disk. All of these programs are installed onto your computer during normal installation.

mboot is the master boot program. This is code that resides in the first 446 bytes of the first sector on the hard disk. Because this sector also contains the partition table for the hard disk, **mboot** is normally written to the hard drive only during installation and only by the **fdisk** utility.

boot, **boot.fha**, and **boot.fva** are variations of the same program. **boot** occupies the first sector of any bootable hard-drive partition. **boot.fha** occupies the first sector of a 5.25-inch, high-density floppy disk. **boot.fva** occupies the first sector of a 3.5-inch, high-density floppy disk.

boot is normally copied to the root partition automatically during installation by a command such as:

```
/bin/dd if=/conf/boot of=/dev/at0a count=1
```

In another example, the following commands format and create a file system on a high-density, 5.25-inch floppy disk:

```
/etc/fdformat -v /dev/fha0
/etc/mkfs /dev/fha0 2400
/bin/cp /conf/boot.fha /dev/fha0
```

When invoked, **boot** loads for the tertiary boot program **tboot**. This, in turn, searches the root directory '/' for file **autoboot**, which is the COHERENT kernel. If it finds this kernel, **boot** loads and invokes it. Otherwise, it gives the prompt ?, and you must type the name of the operating-system kernel to load (typically, "coherent"). If **boot** cannot find the requested kernel or if an error occurs, **boot** does not print an error message, but re-prompts with '?'.

Files

/conf/boot — Boot for AT partitions
/conf/boot.at — Boot for AT partitions (linked to **/conf/boot**)
/conf/boot.atx — AT master boot (linked to **/conf/mboot**)
/conf/boot.f9a — Boot for single-density, nine-sector, 5.25-inch floppy disk
/conf/boot.fha — Boot for 15-sector, 5.25-inch floppy disk
/conf/boot.fqa — Boot for quad-density, nine-sector, 3.5-inch floppy disk

/conf/boot.fva — Boot for 18-sector, 3.5-inch floppy disk
/conf/mboot — AT master boot

See Also

device drivers, fdisk, mboot, mkfs, tboot

boot.fha — Device Driver

Boot block for floppy disk

To be bootable, a COHERENT file system must contain a boot block (either **boot** or **boot.fha**). In addition, all hard disks must contain the master boot block **mboot** or an equivalent.

boot.fha is a boot block for a hard disk partition or a 15-sector floppy. It must be installed as the first sector of the partition or diskette, as follows:

```
/etc/fdformat -a /dev/fha0
/etc/badscan -v -o proto1 /dev/fha0 2400
/etc/mkfs /dev/fha0 proto1
rm proto1
cp /conf/boot.fha /dev/fha0
```

boot.fha searches its root directory '/' for file **autoboot**. If it finds this kernel, **boot.fha** loads and runs it. Otherwise, it gives the prompt ?, to which the user must type the name of the operating-system kernel to load (typically, **coherent**). If **boot.fha** cannot find the requested kernel or if an error occurs, **boot.fha** repeats the prompt and the user must type another name.

Files

/conf/boot.fha — Partition or 15-sector 96tpi floppy boot block

See Also

badscan, boot, device drivers, fdisk, mboot, mkfs

booting — Technical Information

How booting works

Booting is the method by which COHERENT is loaded from a hard disk or floppy disk and set into action. The term comes from the old expression about pulling one's self up by one's bootstraps.

This article discusses the events that take place while booting the COHERENT system. You do not need to read this article to know how to boot COHERENT, as all booting details are handled by COHERENT automatically. However, if you are interested in the details, or want to tailor the system to your needs, it will help.

Two I/O devices are involved in booting. The first device is called the *boot* device; it contains the program necessary to invoke the COHERENT system and start it running. The second device is called the *root* device; it contains the root file system after the system is running. In most cases, these two devices are the same physical device.

Initial Startup

When you boot from a hard disk, your computer's BIOS loads the master boot from the first sector of your hard disk into memory. The master boot then loads the secondary boot from the first sector of your boot partition. When you boot from a floppy disk, however, the BIOS loads the secondary boot directly.

This program, called the *bootstrap* or *secondary boot*, is very small (only 512 bytes), so it cannot do very much. Therefore, its main purpose is to read in a larger, more complex program called the *tertiary boot*, or **/tboot**. It is **/tboot** that actually performs the work of loading the COHERENT system into memory.

If the secondary boot does not find a file called **/tboot**, it print a '?' to prompt for the boot image you want it to load. This indicates a severe error because it means that the tertiary boot can not be found.

If the secondary boot finds **/tboot**, it loads it into memory and lets it take over booting. The first thing **/tboot** does is search for a file called **/autoboot** in the root directory of the device being booted. If **/tboot** finds **/autoboot**, it first pauses for five seconds, so you can abort the process and boot another kernel if you wish. If you do not abort booting within five seconds, **/tboot** then loads **/autoboot** into memory and runs it. If, however, **/tboot** cannot find **/autoboot**, it prompts you to type the name of the COHERENT image to boot, usually **/coherent**. You can type the commands **dir** or **ls** if you do not remember the name of the image you wish to boot. Note that **/autoboot** is usually a link to **/coherent**.

If you need to find the file name of the kernel you are now running (usually **/coherent**), use the program **fifo()**, which is kept in library **libmisc**. See the Lexicon entry **libmisc** for details.

After it loads the system image **/autoboot** from the root device, the system initializes all devices, as well as starting the *idle* process and program **/etc/init**. The idle process uses any leftover computer time.

init controls the operation of the system from this point on. It first executes the command **/etc/brc** (i.e., "boot run commands"), which can run commands like **fsck**. **brc** can request a reboot, remain in single-user mode, or enter multi-user mode automatically. **init** then calls the *shell* to handle commands from the system console. The shell responds by prompting with **#**, and expects regular commands. At this point, the system is in *single-user mode*, which means that no other users can log in to the system. The shell is running in superuser mode and only the console's user is logged in.

At this point, you can enter commands to the system in a normal fashion. One difference from normal, multi-user operation is that the system is in single-user mode, to allow special processing to take place before other users log in. Being in single-user mode gives you the opportunity to run **fsck** to check the file system and perform other administrative tasks before other users log into the system.

When administrative activities are finished, you should type **<ctrl-D>**. This terminates single-user operation; **init** then opens the system to other users.

The file **/etc/rc** contains shell commands that the system executes just before making the system available to other users. This file typically includes commands to delete temporary files and mount standard devices. It also performs any installation-specific commands you require. As system administrator, you maintain this file. You must be sure that it is properly updated and never removed.

One command that must be included in **/etc/rc** is **/etc/update**, which periodically calls **sync()** to update buffered data to the disk.

init also maintains the file **/etc/utmp**, which notes users' login and logout.

Features of the Master Bootstrap

The COHERENT master bootstrap allows you to boot different operating systems from different partitions of any hard drive. It is more powerful than similar programs of other operating systems, and we strongly recommend that you use it. If you do not use the MWC bootstrap, you may have to use floppy disks to boot up MS-DOS and COHERENT. If you have two hard drives and you are placing COHERENT on the second drive, you must use the MWC bootstrap.

The bootstrap can be configured in three ways:

1. No active partition. With this configuration, you have the greatest degree of flexibility. When you boot your system, the following prompt appears on the screen:

   ```
   Select Partition 0-7
   ```

   This means that you must press the number key that corresponds to the partition that holds the root partition of the operating system you wish to boot. (For example, if you wish to boot COHERENT and its root partition is on partition 2, then press the '2' key in response to this prompt.) If you have one hard drive, only partitions 0 through 3 are relevant to you. The bootstrap waits indefinitely until you tell it what to boot.

2. COHERENT is active partition. Under this configuration, the system will automatically boot COHERENT unless you press the number key that represents the root partition of another operating system (e.g., MS-DOS) while the A-drive light is on.

3. MS-DOS (or another operating system) is active partition. Under this configuration, the system automatically boots MS-DOS unless you hit the number key that represents the root partition of another operating system (e.g., COHERENT) while the A-drive light is on.

Under some hardware configurations, particularly faster 80386 machines, having an active partition can cause difficulties when you try to boot a non-active partition. It often is difficult to press the appropriate number key at the right time, and the right time itself can vary. For this reason, the default setting of the master bootstrap is to have no active partition. If at any time you wish to reconfigure the bootstrap, you need only to run the **fdisk** utility under COHERENT and access option 1 (Change active partition) of the option menu. Make the desired change and then save the updated partition table.

Files Used During Startup

The following files are used when the system is in single-user mode:

/etc/drvld.all Device tables to load. This usually names the keyboard driver to use, should you be using the keyboard driver **vtnkb**.

/etc/init Initiate a process on each terminal line, call login when appropriate.

/etc/brc Shell commands for booting.

/etc/checklist List of partitions for **fsck** to check.

/bin/sh Bourne shell.

/bin/ksh Korn shell.

The following files are needed after the system has entered multi-user mode:

/bin/login This file holds the program that controls logging in.

/etc/getty This file holds the executable program that permits a user to log in on a port.

/etc/logmsg This file holds the text of the login prompt.

/etc/motd This file holds the message of the day.

/etc/mount.all Shell script to mount partitions.

/etc/rc This file holds a series of shell commands that **coherent** executes when it enters multi-user startup.

/etc/ttys This file is holds information about terminals. Its contents are read by **getty** to ensure that it sets the port to the correct baud rate and terminal type.

/etc/utmp This file holds information about who is logged in right now. It is read by the command **who**.

Building a Bootable Floppy Disk

Building a bootable floppy disk for COHERENT requires a few more steps than are required to build a bootable floppy for MS-DOS. The task is not particularly painful, it simply requires a little more attention to detail.

The following details the steps required to build a version of COHERENT that can be booted off a floppy disk. Note that the following describes an extremely minimal configuration, which can be used only in single-user mode.

1. Format the Floppy Disk

To begin, format the floppy disk with the command **/etc/fdformat**. After you format the floppy disk, use the command **/etc/mkfs** command to write a blank file system onto it.

2. Write a Bootstrap to the Floppy Disk

To make the floppy disk bootable, you must copy a special program, or *bootstrap*, into the first sector (or *boot block*) of the floppy disk. (This is the same program that is called the *secondary boot* in the above sections.) If a floppy disk is to be bootable, a set of instructions must be present in the boot block that tell the system the name of the kernel — that is, the file on the floppy disk to be loaded and executed.

To write the bootstrap to the floppy disk, you must copy it to the *device* that the floppy disk is in. This ensures that the bootstrap is copied to the first sector, or boot block, of the floppy disk. For example, to copy the bootstrap for a 1.2-megabyte floppy disk in floppy drive 0 (or A), type the command:

```
cp /conf/boot.fha /dev/fha0
```

To copy the bootstrap for a 1.44-megabyte floppy disk to floppy drive 0, type the command:

```
cp /conf/boot.fva /dev/fva0
```

3. Write Tertiary Boot

After you have copied the boot sector, you must mount the floppy device and copy **/tboot** to it. To mount a 1.44-megabyte floppy disk to floppy drive 0, type the command:

```
/etc/mount /dev/fva0 /f0
```

Copy **/tboot** with the following command:

```
cp /tboot /f0
```

Warning: *Never* mount the floppy disk before you copy the bootstrap to it!

See the Lexicon article on **floppy disks** for the table of floppy disk devices to use with the above commands.

4. Copy the Necessary Files

Once the bootstrap is properly written to the floppy disk, it is now time to copy the essential files to it. Type the following commands:

```
mkdir /f0/etc
mkdir /f0/dev
mkdir /f0/bin
mkdir /f0/tmp
cp /tboot /coherent /coherent.sym /f0
cp /etc/init /etc/brc /etc/profile /f0/etc
cp /dev/* /f0/dev
cp /bin/sh /bin/sync /f0/bin
```

If you are using either of the loadable keyboard drivers **nkt** or **vtnkb**, also execute the following commands:

```
mkdir /f0/drv
mkdir /f0/conf
mkdir /f0/conf/kbd
cp /etc/drvld.all /f0/etc
cp /drv/* /f0/drv
cp /conf/kbd/* /f0/conf/kbd
```

The above files will let you run COHERENT in single-user mode, which is all that you need when you boot COHERENT from a floppy disk.

Note that the files **/etc/brc** and **/etc/drvld.all** are scripts that you must modify to suit your needs. The file **/etc/brc** is a key file in the booting process, so be prepared to modify its contents. The significance of this will be reviewed in depth in the next section.

Warning: After you have finished copying files to the floppy disk, execute the command **umount** to unmount the floppy disk. If you do not, the files will be damaged or lost!

5. The Boot Sequence, Modifications To Make the Disk Work

When the computer system powers up and accesses the floppy disk, it reads the boot sector of the disk, which in turn looks for the file **/tboot** and executes it. **/tboot** looks for the kernel named **/autoboot**, reads it, and executes it. If **/tboot** cannot find **/autoboot**, it prompts you to type the name of the kernel to boot.

The kernel loads and invokes **/etc/init** which, in part, looks for and executes the statements in **/etc/brc**, which, in turn, typically loads loadable drivers and runs **/etc/fsck** to check the file systems. If you wish to run **fsck** on the floppy disk, you must copy it from the hard drive.

What is truly important is the *exit status* of **/etc/brc**. If its exit status is not zero, the system remains in single-user mode. If its exit status is zero, the system attempts to enter multiuser mode.

The above-listed files are the bare minimum for a single-user floppy disk. To build a floppy disk with the minimum files needed, your **/etc/brc** file should look like this:

```
/etc/drvld.all
exit 1
```

This forces an exit status of one and causes COHERENT to spawn a single-user shell, **/bin/sh**.

From the shell prompt, you can do whatever you wish, but you are limited to the commands and functions copied to the floppy disk.

/etc/brc is not the only file that may need modification. The kernel (**/coherent** or **/autoboot**) must have the values **rootdev** and **pipedev** patched for the floppy disk's major and minor device numbers. This patching can be done with the commands **/bin/db** or **/conf/patch**.

To patch the kernel on the floppy disk mounted on **/f0** for a 5.25-inch, high-density disk as the root and pipe device, type:

```
/conf/patch /f0/coherent rootdev=makedev\(4,14\)
/conf/patch /f0/coherent pipedev=makedev\(4,14\)
```

For a 3.5-inch, high-density disk, type:

```
/conf/patch /f0/coherent rootdev=makedev\(4,15\)
/conf/patch /f0/coherent pipedev=makedev\(4,15\)
```

Finally, note that when you boot your floppy disk, the disk must *not* be write protected. This is because COHERENT must be able to write temporary files into directory **/tmp**; if it cannot do so, booting will fail.

Uses of a Bootable Floppy Disk

A bootable floppy disk can be a lifesaver should something occur to corrupt the COHERENT file system on the hard drive. A properly prepared floppy can be used to recover a damaged file system by running **/etc/fsck**. You can also use it to copy files from the hard drive should you decide to re-install COHERENT on the hard drive.

Multiuser-mode floppy disks can also be built for the fun of seeing such a system run from a floppy disk. The capacity of such a system is limited, of course, but it can be done.

See Also

Administering COHERENT, boot, libmisc, tboot

Notes

Some users have attempted to use Norton Utilities or similar tools to rearrange the partition table, only to find that COHERENT no longer boots. That is because the kernel has embedded within it the name of the partition on which it and its root file system live. By using Norton Utilities to shuffle the partition table, the kernel will no longer be able to find any of the files or utilities it needs to boot your system.

If you still wish to shuffle your disk's partition table, be sure to change the name of the root device within the kernel *before* you change the partition table.

boottime — System Administration

File that holds time system was last booted

/etc/boottime is an empty file maintained by the **init** process and the **date** command. The modification time of **boottime**, as displayed by the command **ls -l**, is the time that the system was last booted. You can read the time shown by **boottime** with **ls -l**, or with the system calls **stat** or **fstat**.

Files

/etc/boottime

See Also

Administering COHERENT, date, init, mount

Notes

Commands that depend upon **/etc/boottime** may malfunction if the system's date is not set correctly. For instance, the **mount** command depends on the relative modification times of **/etc/boottime** and **/etc/mtab** to detect whether the mount table has been invalidated by a system boot. If the date is set sufficiently far into the past, the mount table may appear to be valid when in fact it is not.

brc — System Administration

Perform maintenance chores, single-user mode
/etc/brc

The shell script **/etc/brc** is executed by the **init** process when the COHERENT system enters single-user mode. The commands in **brc** do such things as set system clock, set the local time zone, and call **fsck** to scan and (if necessary) fix the file systems that that are named in the file **/etc/checklist**.

See Also

Administering COHERENT, checklist, init, rc

Notes

The default message consists of the bell character **<ctrl-G>** plus the text **Going multiuser**. If the bell annoys you, simply delete the **<ctrl-G>** from this string.

break — Command

Exit from shell construct
break [*n*]

The command **break** is used with the shell to control how it performs loops. It is analogous to the **break** keyword in C.

When it is used without an argument, **break** forces the shell to exit from the innermost current **for**, **until**, or **while** loop. If used with an argument, **break** exits from *n* levels of **for**, **until**, or **while** loops.

The shell executes **break** directly.

See Also

commands, continue, for, ksh, sh, until, while

break — C Keyword

Exit from loop or switch statement

break is a C statement that causes an immediate exit from a **switch** sequence, or from a **while**, **for**, or **do** loop.

See Also

C keywords
ANSI Standard, §6.6.6.3

brk() — System Call (libc)

Change size of data area
#include <unistd.h>
brk(*addr***)**
char **addr***;*

The *break* is the lowest address above the data area of a process. **brk()** sets the break to the given *addr*, possibly rounding up by some machine-dependent factor.

See Also

libc, malloc(), sbrk(), unistd.h If the request succeeds, **brk()** returns zero. Otherwise, it returns -1 and sets **errno** to **ENOMEM**.

bsearch() — General Function (libc)

Search an array
#include <stdlib.h>
char ***bsearch(***key, array, number, size, comparison***)**
char ***key*, ****array*;**
size_t *number*, *size*;**
int (*comparison***)();**

bsearch() searches a sorted array for a given item. *item* points to the object sought. *array* points to the base of the array; it has *number* elements, each of which is *size* bytes long. Its elements must be sorted into ascending order before it is searched by **bsearch()**.

comparison points to the function that compares array elements. *comparison* must return zero if its arguments match, a number greater than zero if the element pointed to by *arg1* is greater than the element pointed to by *arg2*, and a number less than zero if the element pointed to by *arg1* is less than the element pointed to by *arg2*.

bsearch() returns a pointer to the array element that matches *item*. If no element matches *item*, then **bsearch()** returns NULL. If more than one element within *array* matches *item*, which element is matched is unspecified.

Example

This example uses **bsearch()** to translate English into "bureaucrat-ese".

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

struct syntab {
      char *english, *bureaucratic;
} cdtab[] = {
/* The left column is in alphabetical order */

      "affect",        "impact",
      "after",         "subsequent to",
      "broke",         "revenue shortfall",
      "building",      "physical facility",
      "call",          "refer to as",
      "do",            "implement",

      "false",         "inoperative",
      "finish",        "finalize",
      "first",         "initial",
      "full",          "in-depth",
      "help",          "facilitate",

      "idiot",         "elected representative",
      "kill",          "terminate with extreme prejudice",
      "lie",           "inoperative statement",
      "order",         "prioritize",
      "talk",          "interpersonal communication",
      "then",          "at that point in time",
      "use",           "utilize"
};

int
comparator(key, item)
char *key;
struct syntab *item;
{
      return(strcmp(key, item->english));
}

main()
{
      struct syntab *ans;
      char buf[80];

      for(;;) {
            printf("Enter an English word: ");
            fflush(stdout);

            if(gets(buf) || !strcmp(buf, "quit") == NULL)
                  break;

            if((ans = bsearch(buf, (char *)cdtab,
                        sizeof(cdtab)/ sizeof(struct syntab),
                        sizeof(struct syntab),
                        comparator)) == NULL)
                  printf("%s not found\n");

            else
                  printf("Don't say \"%s\"; say \"%s\"!\n",
                        ans->english, ans->bureaucratic);
      }

      return(EXIT_SUCCESS);
}
```

See Also

libc, qsort(), stdlib.h
ANSI Standard, §7.10.6.2
POSIX Standard, §8.1

LEXICON

Notes

The name *bsearch* implies that this function performs a binary search. A binary search looks at the midpoint of the array, and compares it with the element being sought. If that element matches, then the work is done. If it does not, then **bsearch()** checks the midpoint of either the upper half of the array or of the lower half, depending upon whether the midpoint of the array is larger or smaller than the item being sought. **bsearch()** bisects smaller and smaller regions of the array until it either finds a match or can bisect no further.

It is important that the input *array* be sorted, or **bsearch()** will not function correctly.

buf.h — Header File

Buffer header
#include <sys/buf.h>

Header file **<sys/buf.h>** defines the structure used to hold buffers.

See Also

header files

buffer — Definition

A *buffer* is a portion of memory set aside to hold data read from or to be written to another process or device. Often, although not always, this involves setting aside a portion of the arena with **malloc** or its related functions.

Buffering, and problems therewith, are encountered most often when using the standard input and output (STDIO) routines. Many operating systems (including COHERENT) automatically place data from a peripheral device into a buffer. Buffers normally can be cleared with **fflush()**, by pressing the carriage return key on routines that perform input, or by sending a newline character on routines that perform output. The function **fclose()**, which closes a file stream, flushes all buffers associated with that stream. **exit()** calls **fclose()**.

Combining unbuffered and buffered I/O functions on the same file or device within one program will produce results that are at best unpredictable.

Example

The following example demonstrates what does and does not happen when you use **fflush()** with the output buffer.

```
#include <stdio.h>
main()
{
      extern char *malloc();
      char *buffer;

      /* use malloc() to create a 120-char buffer */
      if ((buffer = malloc(120)) == NULL) {
            /* if malloc() fails, bail out */
            fprintf(stderr, "malloc failed\n");
            exit(1);
      }

      printf("Type your name:  ");
      fflush(stdout);
      gets(buffer);
      printf("Your name is %s\n", buffer);
}
```

See Also

arena, array, close(), exit(), fflush(), malloc(), Programming COHERENT, stdio.h

build — Command

Install COHERENT onto a hard disk
/etc/build

build installs COHERENT onto your hard disk. COHERENT runs **/etc/build** to install itself onto your hard disk. After installation, you should never have an occasion to run **build**.

See Also

commands

builtin — Command

Execute a command as a built-in command
builtin *command* **[** *arg ...* **]**

The command **ksh** is used by the Korn shell **ksh** to establish *command* as a built-in command.

See Also

commands, ksh

byte — Definition

A **byte** is a group of bits that encodes a character or a small-integer quantity. A byte, like a dollar, consists of eight bits.

The ANSI Standard defines the data type **char** as being equal to one byte. It defines all other data types as multiples of **char**.

See Also

bit, char, data formats, nybble, Programming COHERENT
ANSI Standard, §1.6

byte ordering — Definition

Machine-dependent ordering of bytes

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

```
main()
{
      union
      {
            char b[4];
            int i[2];
            long l;
      } u;
      u.l = 0x12345678L;

      printf("%x %x %x %x\n",
            u.b[0], u.b[1], u.b[2], u.b[3]);
      printf("%x %x\n", u.i[0], u.i[1]);
      printf("%lx\n", u.l);
}
```

When run on "big-endian" machines, such as the M68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

However, when this program is run on "little-endian" machines, such as the PDP-11, you see these results:

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of bytes within words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

C language, canon.h, data formats, Programming COHERENT

bzero() — Sockets Function (libsocket)

Initialize memory to NUL
void bzero(*address***,** *size***)**
char **address***;*
int *size***;**

The function **bzero()** initializes *size* bytes of memory to NUL, beginning at *address*.

See Also

libsocket, memset()

Notes

bzero() is included for compatibility with Berkeley socket code. It is equivalent to the standard C function **memset()**.