object format — Definition

An *object format* describes the form of compiled program that still contains relocation information. The linker **ld** reads file in object format to create executable files.

COHERENT creates object modules that are in the format **l.out**.

See Also

l.out, ld, Programming COHERENT

od — Command

Print an octal dump of a file
od [-bcdox] [*file***] [[+]** *offset***[.][b]]**

od prints the specified *file* as a sequence of octal numbers, or machine words. If no *file* is specified, **od** dumps the standard input.

The following options set the format of **od**'s output:

- **-b** Bytes in default base
- **-c** Bytes in ASCII characters
- **-d** Words in decimal
- **-o** Words in octal
- **-x** Words in hexadecimal

The default base is octal on the PDP-11 and hexadecimal on the i80286, Z-8001, and M68000 families of microprocessors.

Dumping can start at position *offset* into the file. The specified *offset* is octal unless the '.' suffix is present to signify decimal. *offset* is in bytes unless the **b** suffix is present to signify 512-byte blocks.

See Also

ASCII, commands, conv, db, strings

offsetof() — General Macro (stddef.h)

Offset of a field within a structure
#include <stddef.h>
size_t offsetof(*structname***,** *fieldname***);**

offsetof() is a macro that is defined in the header **<stddef.h>**. It returns the number of bytes that the field *fieldname* is offset from the beginning of the the structure *structname*.

offsetof() may return an offset for *fieldname* that is larger than the sum of the sizes of all the members that precede it. This will be due to the fact that some implementations insert padding into a structure to ensure that they are properly aligned.

Example

The following example displays the offset of some fields within a structure:

```
#include <stddef.h>
```

```
struct foo {
      char a[13];
      long b;
      char c[7];
      short d;
      char e[3];
};

main ()
{
      int A, B, C, D, E;

      A = offsetof(struct foo, a[0]);
      B = offsetof(struct foo, b);
      C = offsetof(struct foo, c[0]);
      D = offsetof(struct foo, d);
      E = offsetof(struct foo, e[0]);

      printf ("%d %d %d %d %d\n", A, B, C, D, E);
}
```

When run, this program prints:

```
0 16 20 28 30
```

Note that even though field 'a' of structure **foo** is only 13 bytes long, field 'b' is aligned at byte 16. This is done to conform to the requirements of COFF. For details, see the section on "COFF Linking" in the Lexicon entry for the linker **ld**.

See Also

alignment, C language, ld, libc, stddef.h, struct
ANSI Standard, §7.1.6

open() — System Call (libc)

Open a file
#include <fcntl.h>
int open(*file***,** *type***[,** *mode***])**
char **file***; int** *type***; [int** *mode***;]**

open() opens a *file* to receive data, or to have its data read. When it opens *file*, **open()** returns a file descriptor, which is a small, positive integer that identifies the open *file* for subsequent calls to **read()**, **write()**, **close()**, **dup()**, **dup2()**, or **lseek()**. After *file* is opened, reading or writing begins at byte 0.

The second argument, *type*, determines how the file is opened. It is a bitwise OR of flag bits taken from the following list (as defined in the header file **<fcntl.h>**):

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read and write

One, and only one, of the above three bit values must be set in *flag*. The following bit values can be used to describe further how the file can be opened:

O_NDELAY	Non-blocking I/O
O_APPEND	Append (writes guaranteed at the file's end)
O_SYNC	Sync on every write
O_TRACE	For file system debugging (*non-standard*)
O_NONBLOCK	Non-blocking I/O
O_CREAT	Open with file create (third argument)
O_TRUNC	Open with truncation
O_EXCL	Exclusive open
O_NOCTTY	Do not assign a controlling tty

The remaining bit values are used to how you wish to manipulate *file*:

O_APPEND
Precede every write with an automatic seek to end of *file*.

O_CREAT
If *file* does not exist, create it. If this flag is set the third argument, *mode*, sets the mode on the file. Note that this mode will be masked by **umask()**. See the Lexicon article on the command **chmod** for details on what the various values of *mode* mean.

O_EXCL
Exclusive open: this flag is meaningful only if **O_CREAT** is also used. In that case, **open()** fails with error value **EEXIST** if *file* already exists.

O_NDELAY
No delay in writing to disk. Please note the following caveats when using this flag:

If set: Opening a FIFO with **O_RDONLY** returns without delay. Opening a FIFO with **O_WRONLY** returns an error if no process has the file open for reading. Opening a file associated with a communication line returns without waiting for a carrier signal.

If not set:
Opening a FIFO with **O_RDONLY** blocks until a process opens the file for writing. Opening a FIFO with **O_WRONLY** blocks until a process opens the file for reading. Opening a file associated with a communication line blocks until a carrier signal is present.

O_NOCTTY
If *file* names a terminal device, do not set it to be the controlling terminal for the process.

O_SYNC
All writes to *file* will be synchronous to disk. This means that **write()** will not return until the data have been physically written to disk.

O_TRUNC
If *file* exists, truncate it to zero length. You must have write permissions on *file* to use this flag.

The third argument, *mode*, is significant only if **O_CREAT** is specified in the second argument and if *file* did not exist before the call to **open()**. In that case, *mode* specifies the access permissions of the new file, in exactly the manner that the system call **creat()** uses its *mode* argument to set permissions. The value of *mode* is typically given as either an octal constant or bitwise OR of permission-bit values as defined in header file **<sys/stat.h>**.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]** by using system calls. It demonstrates **open()** plus the system calls **close()**, **read()**, **write()**, and **creat()**.

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

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

main(argc, argv)
int argc; char *argv[];
{
      register int ifd, ofd;
      register unsigned int n;
```

```
      if (argc != 3)
            fatal("Usage: copy source destination");
      if ((ifd = open(argv[1], O_RDONLY)) == -1)
            fatal("cannot open input file");
      if ((ofd = open(argv[2], O_CREAT | O_RDWR | O_TRUNC, 0666)) == -1)
            fatal("cannot open output file");
      /* For COHERENT 286, use creat() instead of open():
       * if ((ofd = creat(argv[2], 0666)) == -1)
       */

      while ((n = read(ifd, buf, BUFSIZE)) != 0) {
            if (n == -1)
                  fatal("read error");
            if (write(ofd, buf, n) != n)
                  fatal("write error");
      }

      if (close(ifd) == -1 || close(ofd) == -1)
            fatal("cannot close");
      exit(0);
}
```

See Also

fopen(), file descriptor, close(), libc
ANSI Standard, §4.9.3
POSIX Standard, §5.3.1

Diagnostics

open() returns -1 if the file does not exist, if the caller lacks permission, or if a system resource is exhausted.

Notes

open() is a low-level call that passes data directly to COHERENT. It should not be mixed with high-level calls, such as **fread()**, **fwrite()**, or **fopen()**.

Code that uses the third argument to **open()** cannot be ported to COHERENT 286.

COHERENT release 4.2.10 changes some of the behaviors triggered by flags **O_EXCL** and **O_NDELAY**. In previous release of COHERENT, flag **O_EXCL** COHERENT would handle blocking subsequent **open()**s. This is no longer the case — the device driver must handle it. In previous release of COHERENT, when flag **O_NDELAY** was used to open a character driver, the I/O flag **IONDLY** would be set. Now, the I/O flag **IONONBLOCK** is set instead.

opendir() — General Function (libc)

Open a directory stream
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (*dirname***)**
char ***dirname*;**

The COHERENT function **opendir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It opens a directory stream and connects the directory *dirname* with it.

opendir() returns a pointer to the directory stream it has created. It returns NULL if it cannot access *dirname*, if *dirname* is not a directory, or if it cannot create the directory stream (perhaps due to insufficient memory).

If an error occurs, **opendir()** exits and sets **errno** to an appropriate value.

Example

The following example searches the current working directory for entry **FOO**:

```
#include <stddef.h>
#include <sys/types.h>
#include <dirent.h>
```

```
main()
{
      DIR *dirp
      struct dirent *dp;

      dirp = opendir( "." );

      while ((dp = readdir( dirp )) != NULL ) {
            if ( strcmp( dp->d_name, "FOO" ) == 0 ) {
                  printf("Found FOO\n");
                  (void) closedir(dirp);
                  return FOUND;
            }
      }

      (void) closedir( dirp );
      printf("FOO not found\n");
      return NOT_FOUND;
}
```

See Also

closedir(), dirent.h, getdents(), libc, readdir(), rewinddir(), seekdir(), telldir()
POSIX Standard, §5.1.2

Notes

The **dirent** routines buffer directories; and because directory entries can appear and disappear as other users manipulate the directory, your application should continually rescan a directory to keep an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

operator — Definition

An **operator** is a function that is built into the C language. It usually relates one operand to another. For example, the statement

```
1+2
```

relates the operands **1** and **2** through the operation of addition; on the other hand, the statement

```
A>B
```

relates the operands **A** and **B** logically, by asserting that the former is greater than the latter; whereas

```
A=B
```

relates the operands **A** and **B** by assigning the value of the latter to the former. The following is a table of the C operators:

*	Multiplication
/	Division
%	Remainder
+	Addition
-	Subtraction
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&&	Logical AND
!=	Inequality
!	Logical negation
||	logical OR
&	Bitwise AND
^	Bitwise exclusive OR
~	Bitwise complement
|	Bitwise inclusive OR

<<	Bitwise shift left
>>	Bitwise shift right
=	Assign
+=	Increment and assign
-=	Decrement and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
++	Increment
--	Decrement
==	Equivalence
&=	Bitwise AND and assign
^=	Bitwise exclusive OR and assign
|=	Bitwise inclusive OR and assign
<<=	Bitwise shift left and assign
>>=	Bitwise shift right and assign
*	Indirection
&	Render an address
()	Function indicator
[]	Array indicator
->	Structure pointer
.	Structure member
? :	Conditional expression
sizeof	size of an object

Precedence

Precedence refers to the order in which C executes operators. The C languages assigns a level of precedence to each operator. Operators are executed in the order of their precedence level, from highest to lowest.

The following table summarizes the precedence of C operators. The are listed in *descending* order of precedence: those listed higher in the table are executed before those lower in the table. Operators listed on the same line have the same level of precedence, and the implementation determines the order in which they are executed. If you use two or more such operators in the same expression, you would be wise to use parentheses to indicate exactly the order in which you want the operators executed.

Operator	*Associativity*
`()  []  ->  .`	Left to right
`!  ~  ++  --  -  (`*type*`)  *  &  sizeof`	Right to left
`*  /  %`	Left to right
`+  -`	Left to right
`<<  >>`	Left to right
`<  <=  >  >=`	Left to right
`==  !=`	Left to right
`&`	Left to right
`^`	Left to right
`|`	Left to right
`&&`	Left to right
`||`	Left to right
`?:`	Right to left
`=  +=  -=  *=  /=  %=`	Right to left
`,`	Left to right

You can always determine precedence in an expression by enclosing sub-expressions within parentheses: the expression enclosed within the innermost parentheses is always executed first.

See Also

Programming COHERENT, sizeof
ANSI Standard, §6.1, §6.3