NAME

file — determine file type

SYNOPSIS

file [**-bchikLnNprsvz**] [**--mime-type**] [**--mime-encoding**] [**-f** *namefile*]
[**-F** *separator*] [**-m** *magicfiles*] *file*
file -C [**-m** *magicfile*]
file [**--help**]

DESCRIPTION

This manual page documents version 5.03 of the **file** command.

file tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: filesystem tests, magic tests, and language tests. The *first* test that succeeds causes the file type to be printed.

The type printed will usually contain one of the words *text* (the file contains only printing characters and a few common control characters and is probably safe to read on an `ASCII` terminal), *executable* (the file contains the result of compiling a program in a form understandable to some `UNIX` kernel or another), or *data* meaning anything else (data is usually 'binary' or non-printable). Exceptions are well-known file formats (core files, tar archives) that are known to contain binary data. When modifying magic files or the program itself, make sure to *preserve these keywords*. Users depend on knowing that all the readable files in a directory have the word 'text' printed. Don't do as Berkeley did and change 'shell commands text' to 'shell script'.

The filesystem tests are based on examining the return from a `stat`(2) system call. The program checks to see if the file is empty, or if it's some sort of special file. Any known file types appropriate to the system you are running on (sockets, symbolic links, or named pipes (FIFOs) on those systems that implement them) are intuited if they are defined in the system header file <`sys/stat.h`>.

The magic tests are used to check for files with data in particular fixed formats. The canonical example of this is a binary executable (compiled program) `a.out` file, whose format is defined in <`elf.h`>, <`a.out.h`> and possibly <`exec.h`> in the standard include directory. These files have a 'magic number' stored in a particular place near the beginning of the file that tells the `UNIX operating system` that the file is a binary executable, and which of several types thereof. The concept of a 'magic' has been applied by extension to data files. Any file with some invariant identifier at a small fixed offset into the file can usually be described in this way. The information identifying these files is read from the compiled magic file `c:/progra~1/file/share/misc/magic.mgc`, or the files in the directory `c:/progra~1/file/share/misc/magic` if the compiled file does not exist. In addition, if `$HOME/.magic.mgc` or `$HOME/.magic` exists, it will be used in preference to the system magic files.

If a file does not match any of the entries in the magic file, it is examined to see if it seems to be a text file. ASCII, ISO-8859-x, non-ISO 8-bit extended-ASCII character sets (such as those used on Macintosh and IBM PC systems), UTF-8-encoded Unicode, UTF-16-encoded Unicode, and EBCDIC character sets can be distinguished by the different ranges and sequences of bytes that constitute printable text in each set. If a file passes any of these tests, its character set is reported. ASCII, ISO-8859-x, UTF-8, and extended-ASCII files are identified as 'text' because they will be mostly readable on nearly any terminal; UTF-16 and EBCDIC are only 'character data' because, while they contain text, it is text that will require translation before it can be read. In addition, **file** will attempt to determine other characteristics of text-type files. If the lines of a file are terminated by CR, CRLF, or NEL, instead of the Unix-standard LF, this will be reported. Files that contain embedded escape sequences or overstriking will also be identified.

Once **file** has determined the character set used in a text-type file, it will attempt to determine in what language the file is written. The language tests look for particular strings (cf. <`names.h`>) that can appear anywhere in the first few blocks of a file. For example, the keyword *.br* indicates that the file is most likely a `troff`(1) input file, just as the keyword *struct* indicates a C program. These tests are less reliable than the previous two groups, so they are performed last. The language test routines also test for some miscellany (such as `tar`(1) archives).

Any file that cannot be identified as having been written in any of the character sets listed above is simply said to be ‘data’.

OPTIONS

-b, **--brief**
Do not prepend filenames to output lines (brief mode).

-c, **--checking-printout**
Cause a checking printout of the parsed form of the magic file. This is usually used in conjunction with the **-m** flag to debug a new magic file before installing it.

-C, **--compile**
Write a `magic.mgc` output file that contains a pre-parsed version of the magic file or directory.

-e, **--exclude** *testname*
Exclude the test named in *testname* from the list of tests made to determine the file type. Valid test names are:

apptype
EMX application type (only on EMX).

text
Various types of text files (this test will try to guess the text encoding, irrespective of the setting of the ‘encoding’ option).

encoding
Different text encodings for soft magic tests.

tokens
Looks for known tokens inside text files.

cdf
Prints details of Compound Document Files.

compress
Checks for, and looks inside, compressed files.

elf
Prints ELF file details.

soft
Consults magic files.

tar
Examines tar files.

-f, **--files-from** *namefile*
Read the names of the files to be examined from *namefile* (one per line) before the argument list. Either *namefile* or at least one filename argument must be present; to test the standard input, use ‘-’ as a filename argument.

-F, **--separator** *separator*
Use the specified string as the separator between the filename and the file result returned. Defaults to ‘:’.

-h, **--no-dereference**
option causes symlinks not to be followed (on systems that support symbolic links). This is the default if the environment variable `POSIXLY_CORRECT` is not defined.

-i, **--mime**
: Causes the file command to output mime type strings rather than the more traditional human readable ones. Thus it may say 'text/plain; charset=us-ascii' rather than 'ASCII text'. In order for this option to work, file changes the way it handles files recognized by the command itself (such as many of the text file types, directories etc), and makes use of an alternative 'magic' file. (See the FILES section, below).

--mime-type, **--mime-encoding**
: Like **-i**, but print only the specified element(s).

-k, **--keep-going**
: Don't stop at the first match, keep going. Subsequent matches will be have the string '\012- ' prepended. (If you want a newline, see the '-r' option.)

-L, **--dereference**
: option causes symlinks to be followed, as the like-named option in `ls`(1) (on systems that support symbolic links). This is the default if the environment variable `POSIXLY_CORRECT` is defined.

-m, **--magic-file** *list*
: Specify an alternate list of files and directories containing magic. This can be a single item, or a colon-separated list. If a compiled magic file is found alongside a file or directory, it will be used instead.

-n, **--no-buffer**
: Force stdout to be flushed after checking each file. This is only useful if checking a list of files. It is intended to be used by programs that want filetype output from a pipe.

-N, **--no-pad**
: Don't pad filenames so that they align in the output.

-p, **--preserve-date**
: On systems that support `utime`(2) or `utimes`(2), attempt to preserve the access time of files analyzed, to pretend that **file** never read them.

-r, **--raw**
: Don't translate unprintable characters to \ooo. Normally **file** translates unprintable characters to their octal representation.

-s, **--special-files**
: Normally, **file** only attempts to read and determine the type of argument files which `stat`(2) reports are ordinary files. This prevents problems, because reading special files may have peculiar consequences. Specifying the **-s** option causes **file** to also read argument files which are block or character special files. This is useful for determining the filesystem types of the data in raw disk partitions, which are block special files. This option also causes **file** to disregard the file size as reported by `stat`(2) since on some systems it reports a zero size for raw disk partitions.

-v, **--version**
: Print the version of the program and exit.

-z, **--uncompress**
: Try to look inside compressed files.

-0, **--print0**
: Output a null character '\0' after the end of the filename. Nice to `cut`(1) the output. This does not affect the separator which is still printed.

--help
: Print a help message and exit.

FILES

`c:/progra~1/file/share/misc/magic.mgc`	Default compiled list of magic.
`c:/progra~1/file/share/misc/magic`	Directory containing default magic files.

ENVIRONMENT

The environment variable `MAGIC` can be used to set the default magic file name. If that variable is set, then **`file`** will not attempt to open `$HOME/.magic`. **`file`** adds '.mgc' to the value of this variable as appropriate. The environment variable `POSIXLY_CORRECT` controls (on systems that support symbolic links), whether **`file`** will attempt to follow symlinks or not. If set, then **`file`** follows symlink, otherwise it does not. This is also controlled by the **`-L`** and **`-h`** options.

SEE ALSO

`magic`(5), `strings`(1), `od`(1), `hexdump`(1,) `file`(1posix)

STANDARDS CONFORMANCE

This program is believed to exceed the System V Interface Definition of FILE(CMD), as near as one can determine from the vague language contained therein. Its behavior is mostly compatible with the System V program of the same name. This version knows more magic, however, so it will produce different (albeit more accurate) output in many cases.

The one significant difference between this version and System V is that this version treats any white space as a delimiter, so that spaces in pattern strings must be escaped. For example,

```
>10     string  language impress       (imPRESS data)
```

in an existing magic file would have to be changed to

```
>10     string  language\ impress      (imPRESS data)
```

In addition, in this version, if a pattern string contains a backslash, it must be escaped. For example

```
0       string          \begindata     Andrew Toolkit document
```

in an existing magic file would have to be changed to

```
0       string          \\begindata    Andrew Toolkit document
```

SunOS releases 3.2 and later from Sun Microsystems include a **`file`** command derived from the System V one, but with some extensions. My version differs from Sun's only in minor ways. It includes the extension of the '&' operator, used as, for example,

```
>16     long&0x7fffffff       >0             not stripped
```

MAGIC DIRECTORY

The magic file entries have been collected from various sources, mainly USENET, and contributed by various authors. Christos Zoulas (address below) will collect additional or corrected magic file entries. A consolidation of magic file entries will be distributed periodically.

The order of entries in the magic file is significant. Depending on what system you are using, the order that they are put together may be incorrect. If your old **`file`** command uses a magic file, keep the old magic file around for comparison purposes (rename it to `c:/progra~1/file/share/misc/magic.orig`).

EXAMPLES

```
$ file file.c file /dev/{wd0a,hda}
file.c:   C program text
file:     ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
          dynamically linked (uses shared libs), stripped
/dev/wd0a: block special (0/0)
/dev/hda: block special (3/0)

$ file -s /dev/wd0{b,d}
```

```
/dev/wd0b: data
/dev/wd0d: x86 boot sector

$ file -s /dev/hda{,1,2,3,4,5,6,7,8,9,10}
/dev/hda:   x86 boot sector
/dev/hda1:  Linux/i386 ext2 filesystem
/dev/hda2:  x86 boot sector
/dev/hda3:  x86 boot sector, extended partition table
/dev/hda4:  Linux/i386 ext2 filesystem
/dev/hda5:  Linux/i386 swap file
/dev/hda6:  Linux/i386 swap file
/dev/hda7:  Linux/i386 swap file
/dev/hda8:  Linux/i386 swap file
/dev/hda9:  empty
/dev/hda10: empty

$ file -i file.c file /dev/{wd0a,hda}
file.c:      text/x-c
file:        application/x-executable
/dev/hda:    application/x-not-regular-file
/dev/wd0a:   application/x-not-regular-file
```

HISTORY

There has been a **file** command in every `UNIX since at least Research Version 4` (man page dated November, 1973). The System V version introduced one significant major change: the external list of magic types. This slowed the program down slightly but made it a lot more flexible.

This program, based on the System V version, was written by Ian Darwin <ian@darwinsys.com> without looking at anybody else's source code.

John Gilmore revised the code extensively, making it better than the first version. Geoff Collyer found several inadequacies and provided some magic file entries. Contributions by the ‘&’ operator by Rob McMahon, cudcv@warwick.ac.uk, 1989.

Guy Harris, guy@netapp.com, made many changes from 1993 to the present.

Primary development and maintenance from 1990 to the present by Christos Zoulas (christos@astron.com).

Altered by Chris Lowth, chris@lowth.com, 2000: Handle the **-i** option to output mime type strings, using an alternative magic file and internal logic.

Altered by Eric Fischer (enf@pobox.com), July, 2000, to identify character codes and attempt to identify the languages of non-ASCII files.

Altered by Reuben Thomas (rrt@sc3d.org), 2007 to 2008, to improve MIME support and merge MIME and non-MIME magic, support directories as well as files of magic, apply many bug fixes and improve the build system.

The list of contributors to the ‘magic’ directory (magic files) is too long to include here. You know who you are; thank you. Many contributors are listed in the source files.

LEGAL NOTICE

Copyright (c) Ian F. Darwin, Toronto, Canada, 1986-1999. Covered by the standard Berkeley Software Distribution copyright; see the file LEGAL.NOTICE in the source distribution.

The files `tar.h` and `is_tar.c` were written by John Gilmore from his public-domain `tar`(1) program, and are not covered by the above license.

BUGS

There must be a better way to automate the construction of the Magic file from all the glop in Magdir. What is it?

file uses several algorithms that favor speed over accuracy, thus it can be misled about the contents of text files.

The support for text files (primarily for programming languages) is simplistic, inefficient and requires recompilation to update.

The list of keywords in `ascmagic` probably belongs in the Magic file. This could be done by using some keyword like '*' for the offset value.

Complain about conflicts in the magic file entries. Make a rule that the magic entries sort based on file offset rather than position within the magic file?

The program should provide a way to give an estimate of 'how good' a guess is. We end up removing guesses (e.g. 'Fromas first 5 chars of file) because' they are not as good as other guesses (e.g. 'Newsgroups:' versus 'Return-Path:'). Still, if the others don't pan out, it should be possible to use the first guess.

This manual page, and particularly this section, is too long.

RETURN CODE

file returns 0 on success, and non-zero on error.

AVAILABILITY

You can obtain the original author's latest version by anonymous FTP on `ftp.astron.com` in the directory `/pub/file/file-X.YZ.tar.gz`

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

file – determine file type

SYNOPSIS

file [-dh][-M *file***][-m** *file***]** *file* **...**

file -i [-h] *file* **...**

DESCRIPTION

The *file* utility shall perform a series of tests in sequence on each specified *file* in an attempt to classify it:

1. If *file* does not exist, cannot be read, or its file status could not be determined, the output shall indicate that the file was processed, but that its type could not be determined.

2. If the file is not a regular file, its file type shall be identified. The file types directory, FIFO, socket, block special, and character special shall be identified as such. Other implementation-defined file types may also be identified. If *file* is a symbolic link, by default the link shall be resolved and *file* shall test the type of file referenced by the symbolic link. (See the **-h** and **-i** options below.)

3. If the length of *file* is zero, it shall be identified as an empty file.

4. The *file* utility shall examine an initial segment of *file* and shall make a guess at identifying its contents based on position-sensitive tests. (The answer is not guaranteed to be correct; see the **-d**, **-M**, and **-m** options below.)

5. The *file* utility shall examine *file* and make a guess at identifying its contents based on context-sensitive default system tests. (The answer is not guaranteed to be correct.)

6. The file shall be identified as a data file.

If *file* does not exist, cannot be read, or its file status could not be determined, the output shall indicate that the file was processed, but that its type could not be determined.

If *file* is a symbolic link, by default the link shall be resolved and *file* shall test the type of file referenced by the symbolic link.

OPTIONS

The *file* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines, except that the order of the **-m**, **-d**, and **-M** options shall be significant.

The following options shall be supported by the implementation:

-d Apply any position-sensitive default system tests and context-sensitive default system tests to the file. This is the default if no **-M** or **-m** option is specified.

-h When a symbolic link is encountered, identify the file as a symbolic link. If **-h** is not specified and *file* is a symbolic link that refers to a nonexistent file, *file* shall identify the file as a symbolic link, as if **-h** had been specified.

-i If a file is a regular file, do not attempt to classify the type of the file further, but identify the file as specified in the STDOUT section.

-M *file* Specify the name of a file containing position-sensitive tests that shall be applied to a file in order to classify it (see the EXTENDED DESCRIPTION). No position-sensitive default system tests nor context-sensitive default system tests shall be applied unless the **-d** option is also specified.

-m *file* Specify the name of a file containing position-sensitive tests that shall be applied to a file in order to classify it (see the EXTENDED DESCRIPTION).

If the **-m** option is specified without specifying the **-d** option or the **-M** option, position-sensitive default system tests shall be applied after the position-sensitive tests specified by the **-m** option. If the **-M** option is specified with the **-d** option, the **-m** option, or both, or the **-m** option is specified with the **-d** option, the concatenation of the position-sensitive tests specified by these options shall be applied in the order specified by the appearance of these options. If a **-M** or **-m** *file* option-argument is **-**, the results are unspecified.

OPERANDS

The following operand shall be supported:

file A pathname of a file to be tested.

STDIN

Not used.

INPUT FILES

The *file* can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *file*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL
If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

NLSPATH
Determine the location of message catalogs for the processing of *LC_MESSAGES* .

ASYNCHRONOUS EVENTS

Default.

STDOUT

In the POSIX locale, the following format shall be used to identify each operand, *file* specified:

"%s: %s\n", <*file*>, <*type*>

The values for <*type*> are unspecified, except that in the POSIX locale, if *file* is identified as one of the types listed in the following table, <*type*> shall contain (but is not limited to) the corresponding string, unless the file is identified by a position-sensitive test specified by a **-M** or **-m** option. Each space shown in the strings shall be exactly one <space>.

Table: File Utility Output Strings

If *file* is:	**<*type*> shall contain the string:**	**Notes**
Nonexistent	cannot open	
Block special	block special	1
Character special	character special	1
Directory	directory	1
FIFO	fifo	1
Socket	socket	1

Symbolic link	symbolic link to	1
Regular file	regular file	1,2
Empty regular file	empty	3
Regular file that cannot be read	cannot open	3
Executable binary	executable	4,6
ar archive library (see *ar*)	archive	4,6
Extended *cpio* format (see *pax*)	cpio archive	4,6
Extended *tar* format (see **ustar** in *pax*)	tar archive	4,6
Shell script	commands text	5,6
C-language source	c program text	5,6
FORTRAN source	fortran program text	5,6
Regular file whose type cannot be determined	data	

Notes:

1. This is a file type test.
2. This test is applied only if the **-i** option is specified.
3. This test is applied only if the **-i** option is not specified.
4. This is a position-sensitive default system test.
5. This is a context-sensitive default system test.
6. Position-sensitive default system tests and context-sensitive default system tests are not applied if the **-M** option is specified unless the **-d** option is also specified.

In the POSIX locale, if *file* is identified as a symbolic link (see the **-h** option), the following alternative output format shall be used:

"%s: %s %s\n", <*file*>, <*type*>, <*contents of link*>"

If the file named by the *file* operand does not exist, cannot be read, or the type of the file named by the *file* operand cannot be determined, this shall not be considered an error that affects the exit status.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

A file specified as an option-argument to the **-m** or **-M** options shall contain one position-sensitive test per line, which shall be applied to the file. If the test succeeds, the message field of the line shall be printed and no further tests shall be applied, with the exception that tests on immediately following lines beginning with a single **'>'** character shall be applied.

Each line shall be composed of the following four <blank>-separated fields:

offset An unsigned number (optionally preceded by a single **'>'** character) specifying the *offset*, in bytes, of the value in the file that is to be compared against the *value* field of the line. If the file is shorter than the specified offset, the test shall fail.

If the *offset* begins with the character **'>'**, the test contained in the line shall not be applied to the file unless the test on the last line for which the *offset* did not begin with a **'>'** was successful. By default, the *offset* shall be interpreted as an unsigned decimal number. With a leading 0x or 0X, the *offset* shall be interpreted as a hexadecimal number; otherwise, with a leading 0, the *offset* shall be interpreted as an octal number.

type The type of the value in the file to be tested. The type shall consist of the type specification characters **c**, **d**, **f**, **s**, and **u**, specifying character, signed decimal, floating point, string, and unsigned decimal, respectively.

The *type* string shall be interpreted as the bytes from the file starting at the specified *offset* and including the same number of bytes specified by the *value* field. If insufficient bytes remain in the file past the *offset* to match the *value* field, the test shall fail.

The type specification characters **d**, **f**, and **u** can be followed by an optional unsigned decimal integer that specifies the number of bytes represented by the type. The type specification character **f** can be followed by an optional **F**, **D**, or **L**, indicating that the value is of type **float**, **double**, or **long double**, respectively. The type specification characters **d** and **u** can be followed by an optional **C**, **S**, **I**, or **L**, indicating that the value is of type **char**, **short**, **int**, or **long**, respectively.

The default number of bytes represented by the type specifiers **d**, **f**, and **u** shall correspond to their respective C-language types as follows. If the system claims conformance to the C-Language Development Utilities option, those specifiers shall correspond to the default sizes used in the *c99* utility. Otherwise, the default sizes shall be implementation-defined.

For the type specifier characters **d** and **u**, the default number of bytes shall correspond to the size of a basic integer type of the implementation. For these specifier characters, the implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **char**, **short**, **int**, or **long**. These numbers can also be specified by an application as the characters **C**, **S**, **I**, and **L**, respectively. The byte order used when interpreting numeric values is implementation-defined, but shall correspond to the order in which a constant of the corresponding type is stored in memory on the system.

For the type specifier **f**, the default number of bytes shall correspond to the number of bytes in the basic double precision floating-point data type of the underlying implementation. The implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **float**, **double**, and **long double**. These numbers can also be specified by an application as the characters **F**, **D**, and **L**, respectively.

All type specifiers, except for **s**, can be followed by a mask specifier of the form *&number*. The mask value shall be AND'ed with the value of the input file before the comparison with the *value* field of the line is made. By default, the mask shall be interpreted as an unsigned decimal number. With a leading 0x or 0X, the mask shall be interpreted as an unsigned hexadecimal number; otherwise, with a leading 0, the mask shall be interpreted as an unsigned octal number.

The strings **byte**, **short**, **long**, and **string** shall also be supported as type fields, being interpreted as **dC**, **dS**, **dL**, and **s**, respectively.

value The *value* to be compared with the value from the file.

If the specifier from the type field is **s** or **string**, then interpret the value as a string. Otherwise, interpret it as a number. If the value is a string, then the test shall succeed only when a string value exactly matches the bytes from the file.

If the *value* is a string, it can contain the following sequences:

\character

The backslash-escape sequences as specified in the Base Definitions volume of IEEE Std 1003.1-2001, Table 5-1, Escape Sequences and Associated Actions (**'\\\\'**, **'\a'**, **'\b'**, **'\f'**, **'\n'**, **'\r'**, **'\t'**, **'\v'**). The results of using any other character, other than an octal digit, following the backslash are unspecified.

\octal

Octal sequences that can be used to represent characters with specific coded values. An octal sequence shall consist of a backslash followed by the longest sequence of one, two, or three octal-digit characters (01234567). If the size of a byte on the system is greater than 9 bits, the valid escape sequence used to represent a byte is implementation-defined.

By default, any value that is not a string shall be interpreted as a signed decimal number. Any such value, with a leading 0x or 0X, shall be interpreted as an unsigned hexadecimal number; otherwise, with a leading zero, the value shall be interpreted as an unsigned octal number.

If the value is not a string, it can be preceded by a character indicating the comparison to be performed. Permissible characters and the comparisons they specify are as follows:

=

The test shall succeed if the value from the file equals the *value* field.

<
The test shall succeed if the value from the file is less than the *value* field.

>
The test shall succeed if the value from the file is greater than the *value* field.

&
The test shall succeed if all of the set bits in the *value* field are set in the value from the file.

^
The test shall succeed if at least one of the set bits in the *value* field is not set in the value from the file.

x
The test shall succeed if the file is large enough to contain a value of the type specified starting at the offset specified.

message
The *message* to be printed if the test succeeds. The *message* shall be interpreted using the notation for the *printf* formatting specification; see *printf*(). If the *value* field was a string, then the value from the file shall be the argument for the *printf* formatting specification; otherwise, the value from the file shall be the argument.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

The following sections are informative.

APPLICATION USAGE

The *file* utility can only be required to guess at many of the file types because only exhaustive testing can determine some types with certainty. For example, binary data on some implementations might match the initial segment of an executable or a *tar* archive.

Note that the table indicates that the output contains the stated string. Systems may add text before or after the string. For executables, as an example, the machine architecture and various facts about how the file was link-edited may be included. Note also that on systems that recognize shell script files starting with **"#!"** as executable files, these may be identified as executable binary files rather than as shell scripts.

EXAMPLES

Determine whether an argument is a binary executable file:

```
file "$1" | grep -Fq executable &&
  printf "%s is executable.\n" "$1"
```

RATIONALE

The **-f** option was omitted because the same effect can (and should) be obtained using the *xargs* utility.

Historical versions of the *file* utility attempt to identify the following types of files: symbolic link, directory, character special, block special, socket, *tar* archive, *cpio* archive, SCCS archive, archive library, empty, *compress* output, *pack* output, binary data, C source, FORTRAN source, assembler source, *nroff*/ *troff*/ *eqn*/ *tbl* source *troff* output, shell script, C shell script, English text, ASCII text, various executables, APL workspace, compiled terminfo entries, and CURSES screen images. Only those types that are reasonably well specified in POSIX or are directly related to POSIX utilities are listed in the table.

Historical systems have used a "magic file" named **/etc/magic** to help identify file types. Because it is generally useful for users and scripts to be able to identify special file types, the **-m** flag and a portable format for user-created magic files has been specified. No requirement is made that an implementation of *file* use this method of identifying files, only that users be permitted to add their own classifying tests.

In addition, three options have been added to historical practice. The **-d** flag has been added to permit users to cause their tests to follow any default system tests. The **-i** flag has been added to permit users to test portably for regular files in shell scripts. The **-M** flag has been added to permit users to ignore any default system tests.

The IEEE Std 1003.1-2001 description of default system tests and the interaction between the **-d**, **-M**, and **-m** options did not clearly indicate that there were two types of "default system tests". The "position-sensitive tests'' determine file types by looking for certain string or binary values at specific offsets in the file being examined. These position-sensitive tests were implemented in historical systems using the magic file described above. Some of these tests are now built into the *file* utility itself on some implementations so the output can provide more detail than can be provided by magic files. For example, a magic file can easily identify a **core** file on most implementations, but cannot name the program file that dropped the core. A magic file could produce output such as:

```
/home/dwc/core: ELF 32-bit MSB core file SPARC Version 1
```

but by building the test into the *file* utility, you could get output such as:

```
/home/dwc/core: ELF 32-bit MSB core file SPARC Version 1, from 'testprog'
```

These extended built-in tests are still to be treated as position-sensitive default system tests even if they are not listed in **/etc/magic** or any other magic file.

The context-sensitive default system tests were always built into the *file* utility. These tests looked for language constructs in text files trying to identify shell scripts, C, FORTRAN, and other computer language source files, and even plain text files. With the addition of the **-m** and **-M** options the distinction between position-sensitive and context-sensitive default system tests became important because the order of testing is important. The context-sensitive system default tests should never be applied before any position-sensitive tests even if the **-d** option is specified before a **-m** option or **-M** option due to the high probability that the context-sensitive system default tests will incorrectly identify arbitrary text files as text files before position-sensitive tests specified by the **-m** or **-M** option would be applied to give a more accurate identification.

Leaving the meaning of **-M -** and **-m -** unspecified allows an existing prototype of these options to continue to work in a backwards-compatible manner. (In that implementation, **-M -** was roughly equivalent to **-d** in IEEE Std 1003.1-2001.)

The historical **-c** option was omitted as not particularly useful to users or portable shell scripts. In addition, a reasonable implementation of the *file* utility would report any errors found each time the magic file is read.

The historical format of the magic file was the same as that specified by the Rationale in the ISO POSIX-2:1993 standard for the *offset*, *value*, and *message* fields; however, it used less precise type fields than the format specified by the current normative text. The new type field values are a superset of the historical ones.

The following is an example magic file:

```
0  short    070707        cpio archive
0  short    0143561       Byte-swapped cpio archive
0  string   070707        ASCII cpio archive
0  long     0177555       Very old archive
0  short    0177545       Old archive
0  short    017437        Old packed data
0  string   \037\036      Packed data
0  string   \377\037      Compacted data
0  string   \037\235      Compressed data
>2 byte&0x80 >0             Block compressed
>2 byte&0x1f x             %d bits
0  string   \032\001      Compiled Terminfo Entry
```

0	**short**	**0433**	**Curses screen image**
0	**short**	**0434**	**Curses screen image**
0	**string**	**<ar>**	**System V Release 1 archive**
0	**string**	**!<arch>\n__.SYMDEF**	**Archive random library**
0	**string**	**!<arch>**	**Archive**
0	**string**	**ARF_BEGARF**	**PHIGS clear text archive**
0	**long**	**0x137A2950**	**Scalable OpenFont binary**
0	**long**	**0x137A2951**	**Encrypted scalable OpenFont binary**

The use of a basic integer data type is intended to allow the implementation to choose a word size commonly used by applications on that architecture.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *ls*, *pax*

COPYRIGHT

NAME

magic_open, **magic_close**, **magic_error**, **magic_file**, **magic_buffer**, **magic_setflags**, **magic_check**, **magic_compile**, **magic_load** — Magic number recognition library.

LIBRARY

Magic Number Recognition Library (libmagic, −lmagic)

SYNOPSIS

#include <magic.h>

magic_t
magic_open(*int flags*);

void
magic_close(*magic_t cookie*);

*const char **
magic_error(*magic_t cookie*);

int
magic_errno(*magic_t cookie*);

*const char **
magic_file(*magic_t cookie, const char *filename*);

*const char **
magic_buffer(*magic_t cookie, const void *buffer, size_t length*);

int
magic_setflags(*magic_t cookie, int flags*);

int
magic_check(*magic_t cookie, const char *filename*);

int
magic_compile(*magic_t cookie, const char *filename*);

int
magic_load(*magic_t cookie, const char *filename*);

DESCRIPTION

These functions operate on the magic database file which is described in magic(5).

The function **magic_open**() creates a magic cookie pointer and returns it. It returns NULL if there was an error allocating the magic cookie. The *flags* argument specifies how the other magic functions should behave:

MAGIC_NONE	No special handling.
MAGIC_DEBUG	Print debugging messages to stderr.
MAGIC_SYMLINK	If the file queried is a symlink, follow it.
MAGIC_COMPRESS	If the file is compressed, unpack it and look at the contents.
MAGIC_DEVICES	If the file is a block or character special device, then open the device and try to look in its contents.
MAGIC_MIME_TYPE	Return a MIME type string, instead of a textual description.
MAGIC_MIME_ENCODING	Return a MIME encoding, instead of a textual description.

`MAGIC_CONTINUE`	Return all matches, not just the first.
`MAGIC_CHECK`	Check the magic database for consistency and print warnings to stderr.
`MAGIC_PRESERVE_ATIME`	On systems that support `utime`(2) or `utimes`(2), attempt to preserve the access time of files analyzed.
`MAGIC_RAW`	Don't translate unprintable characters to a \ooo octal representation.
`MAGIC_ERROR`	Treat operating system errors while trying to open files and follow symlinks as real errors, instead of printing them in the magic buffer.
`MAGIC_NO_CHECK_APPTYPE`	Check for `EMX` application type (only on EMX).
`MAGIC_NO_CHECK_ASCII`	Check for various types of ascii files.
`MAGIC_NO_CHECK_COMPRESS`	Don't look for, or inside compressed files.
`MAGIC_NO_CHECK_ELF`	Don't print elf details.
`MAGIC_NO_CHECK_FORTRAN`	Don't look for fortran sequences inside ascii files.
`MAGIC_NO_CHECK_SOFT`	Don't consult magic files.
`MAGIC_NO_CHECK_TAR`	Don't examine tar files.
`MAGIC_NO_CHECK_TOKENS`	Don't look for known tokens inside ascii files.
`MAGIC_NO_CHECK_TROFF`	Don't look for troff sequences inside ascii files.

The **magic_close**() function closes the `magic`(5) database and deallocates any resources used.

The **magic_error**() function returns a textual explanation of the last error, or NULL if there was no error.

The **magic_errno**() function returns the last operating system error number (`errno`(2)) that was encountered by a system call.

The **magic_file**() function returns a textual description of the contents of the *filename* argument, or NULL if an error occurred. If the *filename* is NULL, then stdin is used.

The **magic_buffer**() function returns a textual description of the contents of the *buffer* argument with *length* bytes size.

The **magic_setflags**() function sets the *flags* described above. Note that using both MIME flags together can also return extra information on the charset.

The **magic_check**() function can be used to check the validity of entries in the colon separated database files passed in as *filename*, or NULL for the default database. It returns 0 on success and -1 on failure.

The **magic_compile**() function can be used to compile the the colon separated list of database files passed in as *filename*, or NULL for the default database. It returns 0 on success and -1 on failure. The compiled files created are named from the `basename`(1) of each file argument with ".mgc" appended to it.

The **magic_load**() function must be used to load the the colon separated list of database files passed in as *filename*, or NULL for the default database file before any magic queries can performed.

The default database file is named by the MAGIC environment variable. If that variable is not set, the default database file name is c:/progra~1/file/share/misc/magic. **magic_load**() adds ".mgc" to the database filename as appropriate.

RETURN VALUES

The function **magic_open**() returns a magic cookie on success and NULL on failure setting errno to an appropriate value. It will set errno to EINVAL if an unsupported value for flags was given. The **magic_load**(), **magic_compile**(), and **magic_check**() functions return 0 on success and -1 on failure. The **magic_file**(), and **magic_buffer**() functions return a string on success and NULL on failure. The **magic_error**() function returns a textual description of the errors of the above functions, or NULL if there was no error. Finally, **magic_setflags**() returns -1 on systems that don't support `utime`(2), or `utimes`(2) when `MAGIC_PRESERVE_ATIME` is set.

FILES

`c:/progra~1/file/share/misc/magic`	The non-compiled default magic database.
`c:/progra~1/file/share/misc/magic.mgc`	The compiled default magic database.

SEE ALSO

`file`(1), `magic`(5)

AUTHORS

Måns Rullgård Initial libmagic implementation, and configuration. Christos Zoulas API cleanup, error code and allocation handling.

NAME

magic — file command's magic pattern file

DESCRIPTION

This manual page documents the format of the magic file as used by the `file`(1) command, version 5.03. The `file`(1) command identifies the type of a file using, among other tests, a test for whether the file contains certain "magic patterns". The file `c:/progra~1/file/share/misc/magic` specifies what patterns are to be tested for, what message or MIME type to print if a particular pattern is found, and additional information to extract from the file.

Each line of the file specifies a test to be performed. A test compares the data starting at a particular offset in the file with a byte value, a string or a numeric value. If the test succeeds, a message is printed. The line consists of the following fields:

`offset` A number specifying the offset, in bytes, into the file of the data which is to be tested.

`type` The type of the data to be tested. The possible values are:

`byte`	A one-byte value.
`short`	A two-byte value in this machine's native byte order.
`long`	A four-byte value in this machine's native byte order.
`quad`	An eight-byte value in this machine's native byte order.
`float`	A 32-bit single precision IEEE floating point number in this machine's native byte order.
`double`	A 64-bit double precision IEEE floating point number in this machine's native byte order.
`string`	A string of bytes. The string type specification can be optionally followed by /[Bbc]*. The "B" flag compacts whitespace in the target, which must contain at least one whitespace character. If the magic has `n` consecutive blanks, the target needs at least `n` consecutive blanks to match. The "b" flag treats every blank in the target as an optional blank. Finally the "c" flag, specifies case insensitive matching: lowercase characters in the magic match both lower and upper case characters in the target, whereas upper case characters in the magic only match uppercase characters in the target.
`pstring`	A Pascal-style string where the first byte is interpreted as the an unsigned length. The string is not NUL terminated.
`date`	A four-byte value interpreted as a UNIX date.
`qdate`	A eight-byte value interpreted as a UNIX date.
`ldate`	A four-byte value interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`qldate`	An eight-byte value interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`beid3`	A 32-bit ID3 length in big-endian byte order.
`beshort`	A two-byte value in big-endian byte order.
`belong`	A four-byte value in big-endian byte order.
`bequad`	An eight-byte value in big-endian byte order.
`befloat`	A 32-bit single precision IEEE floating point number in big-endian byte order.

`bedouble`	A 64-bit double precision IEEE floating point number in big-endian byte order.
`bedate`	A four-byte value in big-endian byte order, interpreted as a Unix date.
`beqdate`	An eight-byte value in big-endian byte order, interpreted as a Unix date.
`beldate`	A four-byte value in big-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`beqldate`	An eight-byte value in big-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`bestring16`	A two-byte unicode (UCS16) string in big-endian byte order.
`leid3`	A 32-bit ID3 length in little-endian byte order.
`leshort`	A two-byte value in little-endian byte order.
`lelong`	A four-byte value in little-endian byte order.
`lequad`	An eight-byte value in little-endian byte order.
`lefloat`	A 32-bit single precision IEEE floating point number in little-endian byte order.
`ledouble`	A 64-bit double precision IEEE floating point number in little-endian byte order.
`ledate`	A four-byte value in little-endian byte order, interpreted as a UNIX date.
`leqdate`	An eight-byte value in little-endian byte order, interpreted as a UNIX date.
`leldate`	A four-byte value in little-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`leqldate`	An eight-byte value in little-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`lestring16`	A two-byte unicode (UCS16) string in little-endian byte order.
`melong`	A four-byte value in middle-endian (PDP-11) byte order.
`medate`	A four-byte value in middle-endian (PDP-11) byte order, interpreted as a UNIX date.
`meldate`	A four-byte value in middle-endian (PDP-11) byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
`indirect`	Starting at the given offset, consult the magic database again.
`regex`	A regular expression match in extended POSIX regular expression syntax (like egrep). Regular expressions can take exponential time to process, and their performance is hard to predict, so their use is discouraged. When used in production environments, their performance should be carefully checked. The type specification can be optionally followed by `/[c][s]`. The “c” flag makes the match case insensitive, while the “s” flag update the offset to the start offset of the match, rather than the end. The regular expression is tested against line `N + 1` onwards, where `N` is the given offset. Line endings are assumed to be in the machine’s native format. `^` and `$` match the beginning and end of individual lines, respectively, not beginning and end of file.
`search`	A literal string search starting at the given offset. The same modifier flags can be used as for string patterns. The modifier flags (if any) must be followed by `/number` the range, that is, the number of positions at which the match will be attempted, starting from the start offset. This is suitable for

searching larger binary expressions with variable offsets, using \ escapes for special characters. The offset works as for regex.

`default` This is intended to be used with the test *x* (which is always true) and a message that is to be used if there are no other matches.

Each top-level magic pattern (see below for an explanation of levels) is classified as text or binary according to the types used. Types “regex” and “search” are classified as text tests, unless non-printable characters are used in the pattern. All other tests are classified as binary. A top-level pattern is considered to be a test text when all its patterns are text patterns; otherwise, it is considered to be a binary pattern. When matching a file, binary patterns are tried first; if no match is found, and the file looks like text, then its encoding is determined and the text patterns are tried.

The numeric types may optionally be followed by `&` and a numeric value, to specify that the value is to be AND’ed with the numeric value before any comparisons are done. Prepending a `u` to the type indicates that ordered comparisons should be unsigned.

`test` The value to be compared with the value from the file. If the type is numeric, this value is specified in C form; if it is a string, it is specified as a C string with the usual escapes permitted (e.g. \n for new-line).

Numeric values may be preceded by a character indicating the operation to be performed. It may be =, to specify that the value from the file must equal the specified value, <, to specify that the value from the file must be less than the specified value, >, to specify that the value from the file must be greater than the specified value, `&`, to specify that the value from the file must have set all of the bits that are set in the specified value, `^`, to specify that the value from the file must have clear any of the bits that are set in the specified value, or `~`, the value specified after is negated before tested. `x`, to specify that any value will match. If the character is omitted, it is assumed to be =. Operators `&`, `^`, and `~` don’t work with floats and doubles. The operator `!` specifies that the line matches if the test does *not* succeed.

Numeric values are specified in C form; e.g. `13` is decimal, `013` is octal, and `0x13` is hexadecimal.

For string values, the string from the file must match the specified string. The operators =, < and > (but not `&`) can be applied to strings. The length used for matching is that of the string argument in the magic file. This means that a line can match any non-empty string (usually used to then print the string), with >*\0* (because all non-empty strings are greater than the empty string).

The special test *x* always evaluates to true. `message` The message to be printed if the comparison succeeds. If the string contains a `printf`(3) format specification, the value from the file (with any specified masking performed) is printed using the message as the format string. If the string begins with “\b”, the message printed is the remainder of the string with no whitespace added before it: multiple matches are normally separated by a single space.

An APPLE 4+4 character APPLE creator and type can be specified as:

```
!:apple CREATYPE
```

A MIME type is given on a separate line, which must be the next non-blank or comment line after the magic line that identifies the file type, and has the following format:

```
!:mime  MIMETYPE
```

i.e. the literal string “!:mime” followed by the MIME type.

An optional strength can be supplied on a separate line which refers to the current magic description using the following format:

```
!:strength OP VALUE
```

The operand `OP` can be: `+`, `−`, `*`, or `/` and `VALUE` is a constant between 0 and 255. This constant is applied using the specified operand to the currently computed default magic strength.

Some file formats contain additional information which is to be printed along with the file type or need additional tests to determine the true file type. These additional tests are introduced by one or more > characters preceding the offset. The number of > on the line indicates the level of the test; a line with no > at the beginning is considered to be at level 0. Tests are arranged in a tree-like hierarchy: If a the test on a line at level *n* succeeds, all following tests at level *n+1* are performed, and the messages printed if the tests succeed, untile a line with level *n* (or less) appears. For more complex files, one can use empty messages to get just the "if/then" effect, in the following way:

```
0      string   MZ
>0x18  leshort  <0x40   MS-DOS executable
>0x18  leshort  >0x3f   extended PC executable (e.g., MS Windows)
```

Offsets do not need to be constant, but can also be read from the file being examined. If the first character following the last > is a (then the string after the parenthesis is interpreted as an indirect offset. That means that the number after the parenthesis is used as an offset in the file. The value at that offset is read, and is used again as an offset in the file. Indirect offsets are of the form: *((x [.[bislBISL]][+−][y])*. The value of *x* is used as an offset in the file. A byte, id3 length, short or long is read at that offset depending on the *[bislBISLm]* type specifier. The capitalized types interpret the number as a big endian value, whereas the small letter versions interpret the number as a little endian value; the *m* type interprets the number as a middle endian (PDP-11) value. To that number the value of *y* is added and the result is used as an offset in the file. The default type if one is not specified is long.

That way variable length structures can be examined:

```
# MS Windows executables are also valid MS-DOS executables
0           string  MZ
>0x18       leshort <0x40   MZ executable (MS-DOS)
# skip the whole block below if it is not an extended executable
>0x18       leshort >0x3f
>>(0x3c.l)  string  PE\0\0  PE executable (MS-Windows)
>>(0x3c.l)  string  LX\0\0  LX executable (OS/2)
```

This strategy of examining has a drawback: You must make sure that you eventually print something, or users may get empty output (like, when there is neither PE\0\0 nor LE\0\0 in the above example)

If this indirect offset cannot be used directly, simple calculations are possible: appending *[+-*/%&|^]number* inside parentheses allows one to modify the value read from the file before it is used as an offset:

```
# MS Windows executables are also valid MS-DOS executables
0           string  MZ
# sometimes, the value at 0x18 is less that 0x40 but there's still an
# extended executable, simply appended to the file
>0x18       leshort <0x40
>>(4.s*512) leshort 0x014c  COFF executable (MS-DOS, DJGPP)
>>(4.s*512) leshort !0x014c MZ executable (MS-DOS)
```

Sometimes you do not know the exact offset as this depends on the length or position (when indirection was used before) of preceding fields. You can specify an offset relative to the end of the last up-level field using ‘&’ as a prefix to the offset:

```
0           string  MZ
>0x18       leshort >0x3f
>>(0x3c.l)  string  PE\0\0    PE executable (MS-Windows)
# immediately following the PE signature is the CPU type
>>>&0       leshort 0x14c     for Intel 80386
>>>&0       leshort 0x184     for DEC Alpha
```

Indirect and relative offsets can be combined:

```
0              string  MZ
>0x18          leshort <0x40
>>(4.s*512)    leshort !0x014c MZ executable (MS-DOS)
# if it's not COFF, go back 512 bytes and add the offset taken
# from byte 2/3, which is yet another way of finding the start
# of the extended executable
>>>&(2.s-514) string  LE      LE executable (MS Windows VxD driver)
```

Or the other way around:

```
0                  string  MZ
>0x18              leshort >0x3f
>>(0x3c.l)         string  LE\0\0  LE executable (MS-Windows)
# at offset 0x80 (-4, since relative offsets start at the end
# of the up-level match) inside the LE header, we find the absolute
# offset to the code area, where we look for a specific signature
>>>(&0x7c.l+0x26) string  UPX     \b, UPX compressed
```

Or even both!

```
0                 string  MZ
>0x18             leshort >0x3f
>>(0x3c.l)        string  LE\0\0 LE executable (MS-Windows)
# at offset 0x58 inside the LE header, we find the relative offset
# to a data area where we look for a specific signature
>>>&(&0x54.l-3)  string  UNACE  \b, ACE self-extracting archive
```

Finally, if you have to deal with offset/length pairs in your file, even the second value in a parenthesized expression can be taken from the file itself, using another set of parentheses. Note that this additional indirect offset is always relative to the start of the main indirect offset.

```
0                  string       MZ
>0x18              leshort      >0x3f
>>(0x3c.l)         string       PE\0\0 PE executable (MS-Windows)
# search for the PE section called ".idata"...
>>>&0xf4           search/0x140 .idata
# ...and go to the end of it, calculated from start+length;
# these are located 14 and 10 bytes after the section name
>>>>(&0xe.l+(-4)) string       PK\3\4 \b, ZIP self-extracting archive
```

SEE ALSO

`file`(1) – the command that reads this file.

BUGS

The formats `long`, `belong`, `lelong`, `melong`, `short`, `beshort`, `leshort`, `date`, `bedate`, `medate`, `ledate`, `beldate`, `leldate`, and `meldate` are system-dependent; perhaps they should be specified as a number of bytes (2B, 4B, etc), since the files being recognized typically come from a system on which the lengths are invariant.