

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 5.22. 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 `/usr/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 `/[WwcCtbT]*`. The “W” flag compacts whitespace in the target, which must contain at least one whitespace character. If the magic hasn consecutive e blanks, the target needs at least `n` consecutive blanks to match. The “w” flag treats every blank in the magic as an optional blank. The “c” flag specifies case insensitive matching: lower case characters in the magic match both lower and upper case characters in the target, whereas upper case characters in the magic only match upper case characters in the target. The “C” flag specifies case insensitive matching: upper case characters in the magic match both lower and upper case characters in the target, whereas lower case characters in the magic only match upper case characters in the target. To do a complete case insensitive match, specify both “c” and “C”. The “t” flag forces the test to be done for text files, while the “b” flag forces the test to be done for binary files. The “T” flag causes the string to be trimmed, i.e. leading and trailing whitespace is deleted before the string is printed.
 - pstring** A Pascal-style string where the first byte/short/int is interpreted as the unsigned length. The length defaults to byte and can be specified as a modifier. The following modifiers are supported:
 - B** A byte length (default).
 - H** A 2 byte big endian length.

	h	A 2 byte big little length.
	L	A 4 byte big endian length.
	l	A 4 byte big little length.
	J	The length includes itself in its count. The string is not NUL terminated. “J” is used rather than the more valuable “T” because this type of length is a feature of the JPEG format.
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.
qwdate		An eight-byte value interpreted as a Windows-style date.
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.
beqwdate		An eight-byte value in big-endian byte order, interpreted as a Windows-style date.
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.

<code>ledouble</code>	A 64-bit double precision IEEE floating point number in little-endian byte order.
<code>ledate</code>	A four-byte value in little-endian byte order, interpreted as a UNIX date.
<code>leqdate</code>	An eight-byte value in little-endian byte order, interpreted as a UNIX date.
<code>leldate</code>	A four-byte value in little-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
<code>leqldate</code>	An eight-byte value in little-endian byte order, interpreted as a UNIX-style date, but interpreted as local time rather than UTC.
<code>leqwdate</code>	An eight-byte value in little-endian byte order, interpreted as a Windows-style date.
<code>lestring16</code>	A two-byte unicode (UCS16) string in little-endian byte order.
<code>melong</code>	A four-byte value in middle-endian (PDP-11) byte order.
<code>medate</code>	A four-byte value in middle-endian (PDP-11) byte order, interpreted as a UNIX date.
<code>meldate</code>	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.
<code>indirect</code>	Starting at the given offset, consult the magic database again. The offset of the <code>indirect</code> magic is by default absolute in the file, but one can specify <code>/r</code> to indicate that the offset is relative from the beginning of the entry.
<code>name</code>	Define a “named” magic instance that can be called from another <code>use</code> magic entry, like a subroutine call. Named instance direct magic offsets are relative to the offset of the previous matched entry, but indirect offsets are relative to the beginning of the file as usual. Named magic entries always match.
<code>use</code>	Recursively call the named magic starting from the current offset. If the name of the referenced begins with a <code>~</code> then the endianness of the magic is switched; if the magic mentioned <code>leshort</code> for example, it is treated as <code>beshort</code> and vice versa. This is useful to avoid duplicating the rules for different endianness.
<code>regex</code>	A regular expression match in extended POSIX regular expression syntax (like <code>egrep</code>). 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 size of the string to search should also be limited by specifying <code>/<length></code> , to avoid performance issues scanning long files. The type specification can also be optionally followed by <code>/[c][s][l]</code> . The “ <code>c</code> ” flag makes the match case insensitive, while the “ <code>s</code> ” flag update the offset to the start offset of the match, rather than the end. The “ <code>l</code> ” modifier, changes the limit of length to mean number of lines instead of a byte count. Lines are delimited by the platforms native line delimiter. When a line count is specified, an implicit byte count also computed assuming each line is 80 characters long. If neither a byte or line count is specified, the search is limited automatically to 8KiB. <code>^</code> and <code>\$</code> 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 search expression must contain the range in the form `/number`, 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 order of modifier and number is not relevant.
- default** This is intended to be used with the test `x` (which is always true) and it has no type. It matches when no other test at that continuation level has matched before. Clearing that matched tests for a continuation level, can be done using the `clear` test.
- clear** This test is always true and clears the match flag for that continuation level. It is intended to be used with the `default` test.

For compatibility with the Single UNIX Standard, the type specifiers `dC` and `d1` are equivalent to `byte`, the type specifiers `uC` and `u1` are equivalent to `ubyte`, the type specifiers `dS` and `d2` are equivalent to `short`, the type specifiers `uS` and `u2` are equivalent to `ushort`, the type specifiers `dI`, `dL`, and `d4` are equivalent to `long`, the type specifiers `uI`, `uL`, and `u4` are equivalent to `ulong`, the type specifier `d8` is equivalent to `quad`, the type specifier `u8` is equivalent to `uquad`, and the type specifier `s` is equivalent to `string`. In addition, the type specifier `dQ` is equivalent to `quad` and the type specifier `uQ` is equivalent to `uquad`.

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 an `y` 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.

Numeric operations are not performed on date types, instead the numeric value is interpreted as an offset.

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).

Dates are treated as numerical values in the respective internal representation.

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 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, until 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.1) string PE\0\0 PE executable (MS-Windows)
>>(0x3c.1) 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 PE00 nor LE00 in the above example)

If this indirect offset cannot be used directly, simple calculations are possible: appending `[+-%E!^]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 than 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.1) 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.1) 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.1+0x26) string UPX \b, UPX compressed
```

Or even both!

```
0 string MZ
>0x18 leshort >0x3f
>>(0x3c.1) 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
>>>&(&0x54.1-3) string UNACE \b, ACE self-extracting archive
```

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.1) 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.1+(-4)) string PK\3\4 \b, ZIP self-extracting archive
```

If you have a list of known values at a particular continuation level, and you want to provide a switch-like default case:

```
# clear that continuation level match
>18 clear
>18 lelong 1 one
>18 lelong 2 two
>18 default x
# print default match
>>18 lelong x unmatched 0x%x
```

SEE ALSO

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

BUGS

The formats `long`, `belong`, `lelong`, `melong`, `short`, `beshort`, and `leshort` do not depend on the length of the C data types `short` and `long` on the platform, even though the Single UNIX Specification implies that they do. However, as OS X Mountain Lion has passed the Single UNIX Specification validation suite, and supplies a version of `file(1)` in which they do not depend on the sizes of the C data types and that is built for a 64-bit environment in which `long` is 8 bytes rather than 4 bytes, presumably the validation suite does not test whether, for example `long` refers to an item with the same size as the C data type `long`. There should probably be type names `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`, and specified-byte-order variants of them, to make it clearer that those types have specified widths.