NAME

encodef − encode/decode filenames or similar data

SYNOPSIS

encodef [*options*...] [-- [*filenames*...]]

decodef [*options*...] [-- [*filenames*...]]

DESCRIPTION

POSIX/Unix/Linux filenames and pathnames can include nearly-arbitrary sequences of bytes (octets), including newlines, tabs, terminal control sequences, spaces, sequences that are not legal in the current locale's encoding, and so on. Many text-processing utilities cannot easily process arbitrary filenames directly, e.g., they may misinterpret a newline in the middle of a filename as the end of the filename. Some tools can generate or accept filenames terminated by a null byte, but many text-processing utilities cannot easily process these either.

The *encodef* and *decodef* utilities make it much easier to process arbitrary filenames. The *encodef* utility accepts filenames and sends to stdout encoded filenames that are much easier to process. The *decodef* utility reverses this; it accepts encoded filenames and sends to stdout the decoded filenames that can be directly used. These utilities can be used for arbitrary data, not just filenames, but they are specifically designed to work well with filenames.

More specifically, the *encodef* utility accepts 0 or more filenames from its command line if an "--" end-of-options argument is provided; or it reads from stdin instead if no "--" is provided. It sends to stdout the encoding of each filename, in order, with each encoded filename terminated by a newline. If stdin is read, the filenames are separated by a null byte (newline cannot be used since filenames can include newlines). The last filename from stdin may, but need not, end with a null byte. This processing of stdin enables easy processing or transition from filename lists where filenames are terminated or separated by a null byte. By default, *encodef* uses percent encoding (e.g., %hh), though other encodings are available.

When encoding, the tool must determine which bytes (characters) are encoded. By default, *encodef* encodes most characters except A-Z, a-z, 0-9, <slash> (/), <period> (.), <underscore> (_), and non-leading <hyphen> (-). A dash is leading if it is at the beginning of a name or immediately follows a forward slash. This encodes most characters, and is the default because it results in the safest possible filenames, reducing the risk of errors or security vulnerabilities due to mishandled filenames. In particular, this default escapes: (1) the control characters including newline, tab, and terminal control characters, (2) the space character, so shell users can use the default IFS setting, and (3) the shell globbing characters * and ?. In addition, by default all of the unescaped characters and the escape characters are in the POSIX "portable character set", so they *must* exist in any given locale. (This also means that all other letters are encoded, but this is often a good thing; there is no way to be certain of a filename's character encoding in the general case.) Various options (see -C and -S) control which characters are encoded.

The decodef utility reverses *encodef*; the encodef -d (decode) option makes encodef perform the work of decodef. It accepts encoded filenames from the command line (or from stdin if no "--" option is provided) and decodes the filenames back to their original form, sending the result to stdout. If only a single filename is provided, the output on stdout is simply the undecoded name. When reading from stdin, each filename is terminated by newline or null byte; the last filename need not be terminated. By default, if multiple filenames are provided, the output on stdout is the undecoded names separated by a null byte (since this is the only value that cannot occur in a filename; the last value is not terminated with a null byte). One odd option, -Y, appends the letter Y to the end of each filename, and is needed to properly decode filenames using shell command substitution (as explained below).

Several different filename encodings are supported:

- -U: URI encoding, aka URL encoding aka percent encoding. In this encoding, the percent character % is the escape character, and %hh represents the byte with the hexadecimal value hh (in either upper or lower case). Either %% or %25 is accepted as an encoding of the percent character %, but %25 is what is generated when encoding %. This encoding is described and used in IETF RFC 3986. This

particular option flag was selected because -U is not used by GNU xargs, and %U is not used in GNU printf(1); GNU is both common and very feature-rich, so it seemed likely that these flag values wouldn't be used by others. Unfortunately, there don't seem to be many flags that aren't already used by find's -printf; GNU find -printf already uses %p, %P, %u, and %U for other things.

- -B: traditional backslash encoding. In this encoding, the backslash character \ is the escape character. The expression \ddd, where ddd is a 1 to 3 digit octal value, can represent an arbitrary byte; note that a leading 0 is not required, so byte 255 is represented as \377, and not \0377 as with printf(1)'s %b. The usual escape sequences are also accepted from POSIX: '\\' (for backslash itself), '\a', '\b', '\f', '\n', '\r', '\t', and '\v'. When decoding, a backslash followed a non-alphanumeric character is replaced by that character, so \" encodes ". As an extension, this decoder also accepts \xHH as a representation of the hexadecimal number HH in upper or lower case (the encoder does not generate this, but some other programs accept this representation; localedef(1) supports this notation, for example). This encoding for backslash is used by printf(1) format strings, printf(3), C, C++, tr(1), and many other tools. However, beware of providing encoded filenames as the format string values to most such tools directly, because they will often interpret other characters such as '%' in a way that will cause them to fail (and in some cases can lead to a security vulnerability). The *decode* utility does *not* interpret other characters like '%' specially, and thus does not have this risk. This is *not* the encoding used by printf %b (aka "pfb" or "extra 0" encoding). This particular option flag was selected because -B is not used by GNU xargs, and %B is not used in GNU printf(1) (%b is part of the POSIX standard, but that's different). GNU find -printf does not use %B, which is nice (so that COULD be used).
- -b (extra 0): printf(1) %b (pfb) encoding. This is the format decoded by POSIX's printf(1) %b format (but *not* by the printf(1) format string itself, nor the format used by other POSIX tools). In this encoding, the backslash character \ is the escape character. The expression \0ddd, where ddd is a 0 to 3 digit octal value, can represent an arbitrary byte; note that a leading "extra" 0 is always required with octal codes, so byte 255 is represented as \0377 and not \377. It also accepts and generates the POSIX encodings '\\' (for backslash itself), '\a', '\b', '\f', '\n', '\r', '\t', and '\v'. Note that printf(1) %b does *not* portably accept some other common escape patterns, such as \" or \' or \ followed by a space. This is an odd encoding, and not what many people think it is. Its main advantages are that it is portable (because it is part of POSIX.1-2008 printf(1)), widely implemented, and often efficient in shells (because printf(1) is typically a shell built-in). This particular option flag was selected because -b is not used by GNU xargs, and %b is the POSIX-required name for the relevant printf(1). Unfortunately, GNU find -printf already uses %b for something else. (Older versions of this tool use -e for "extra 0".)

Note that for purposes of these utilities, encodings have little to do with the character encoding used for internationalization. In POSIX, filenames are simply sequences of bytes; there is no standard mechanism to determine what character encoding was used to encode a given filename. The author recommends that users always use UTF-8 as the character encoding to encode all filenames.

OPTIONS

Options can control factors such as which encoding scheme is used (see -B, -b, -U), which bytes will be encoded (see -C and -S), and the format of the results (see -Y). The following options are supported:

−B Use traditional backslash encoding. The expression \ddd, where ddd is a 1 to 3 digit octal value, represent an arbitrary byte. The encoder produces, and the decoder accepts, the usual expressions '\\' (for backslash itself), '\a', '\b', '\f', '\n', '\r', '\t', and '\v'. The decoder also accepts '\"' (for double quote), '\'' (for single quote), and '\ ' (for space). As an extension, this decoder also accepts \xHH as a representation of the hexadecimal number HH in upper or lower case (the encoder does not generate this, but some other programs accept this representation). When encoding, this is the encoding of the *output*; when decoding, this is the encoding of the *input*.

−d Decode. If passed as an option to encodef, makes it work as decodef.

−b Use the "extra 0" (aka printf %b or pfb) encoding. This is the encoding accepted by the printf %b format specifier, though it is not the same as many other POSIX tools. The expression \0ddd, where ddd is a 0 to 3 digit octal value, can represent an arbitrary byte; note that a leading "extra" 0

is always required with octal codes. It also accepts and generates the POSIX encodings '\\' (for backslash itself), '\a', '\b', '\f', '\n', '\r', '\t', and '\v'. Note that printf %b is not guaranteed to accept \" and other escape sequences. When encoding, this is the encoding of the *output*; when decoding, this is the encoding of the *input*.

−C (Encoder only) Only encode control characters and the "escape" character (\ or %). It also encodes "-" if it is at the beginning of a pathname component, to reduce the risk from leading "-" (these are often confused with option flags). See also -S.

−S (Encoder only) Like -C, but also encode the byte representing the space character. This is useful when handing filenames to the shell; the default IFS values (space, tab, newline) are all encoded. See also -C.

−U Use percent encoding (aka URL encoding), where %HH represents the hexadecimal value HH. When encoding, this is the encoding of the *output*; when decoding, this is the encoding of the *input*.

-Y (Decoder only) Y-append mode. Append the letter 'Y' after the end of each filename in the output. This unusual mode supports shell scripts. Shell command substitution strips off trailing newlines, which could corrupt filenames that end with newline. If a shell script uses command substitution to invoke the decoder directly, it should use this option to ensure that the last character is always 'Y' and then strip off the trailing 'Y'. ("X" is not used as the trailing letter; it's being reserved for options involving xargs and XML.)

EXIT STATUS
ENVIRONMENT
FILES
CONFORMING TO

Percent encoding is defined in IETF RFC 3986. POSIX.1-2008 defines both of the backslash quotation systems supported by printf(1); in particular, POSIX.1-2008 XBD Chapter 5 table 5-1, page 121 defines '\\' (for backslash itself), '\a', '\b', '\f', '\n', '\r', '\t', and '\v'.

NOTES

The encoder and decoder can process arbitrary filename lengths, but the underlying filesystem, interfaces, and utilities will have some sort of limit.

BUGS

The fact that POSIX systems permit filenames to contain control characters like newline, return, tab, and ESC (used for terminal escapes) could be considered a bug all by itself; forbidding them would reduce the need for this pair of tools. If POSIX systems could never encounter bytes 1 through 31 in filenames, filename processing would be *much* simpler. For example, once a shell script sets IFS to newline and tab (e.g., near its beginning):

```
IFS="‘printf '\n\t'‘"
```

It could loop over all filenames in the current tree with the much simpler expression:

```
for f in ‘find .‘ ; do ...
```

POSIX does not require that filenames meet a particular character encoding such as UTF-8; we recommend that users always use UTF-8, to simplify worldwide exchange of data.

Currently this doesn't have options to control which bytes will be encoded; I'm sure that will change.

EXAMPLE

You can display the filenames from the current directory down, encoded in percent encoding, with:

```
find . -exec encodef -p -- {} \+
```

You can loop over arbitrary trees of files in shell with encodef. Here is one example, which uses -e encoding (which is supported by the typically built-in printf(1)). Since newline, tab, and space are all encoded by default, you can use the default value of IFS and still have correct filenames. Note the "trailing Y" trick in the second line; filenames can end in newline, and command substitution strips off trailing newlines, so you need to something like the following if filenames might end with a newline:

```
for ef in ‘find . -exec encodef -e -- {} \+‘ ; do
  filename="$(printf "%bY" "$ef")" ; filename="${filename%Y}" ...
```

Here is a similar loop, but using the percent encoding, and showing how to use decodef. Note that when decoding filenames via shell command substitution, you should normally use -Y or filenames with trailing newlines will be corrupted:

```
for ef in ‘find . -exec encodef -p -- {} \+‘ ; do
  filename="$(decodef -pY -- "$ef")" ; filename="${filename%Y}" ...
```

You can use the "xargsf" utility to simultaneously decode the filename and run a program using the filename; this is sometimes easier. The "xargsf" utility is simply a variant of xargs that decodes the filenames provided (per decodef), and then runs the indicated program(s) on those filenames. (In the future, it’s hoped that xargs would add the basic features of decodef, but having a separate "xargsf" lets people easily experiment.) Here is an example:

```
for f in ‘find . -exec encodef -p -- {} \+‘ ; do
  printf "%s" "$f" | xargsf -p ls -l
```

AUTHOR

Written by David A. Wheeler.

SEE ALSO

printf(1), iconv(1), sh(1), find(1), xargs(1).

See its web page http://www.dwheeler.com/encodef