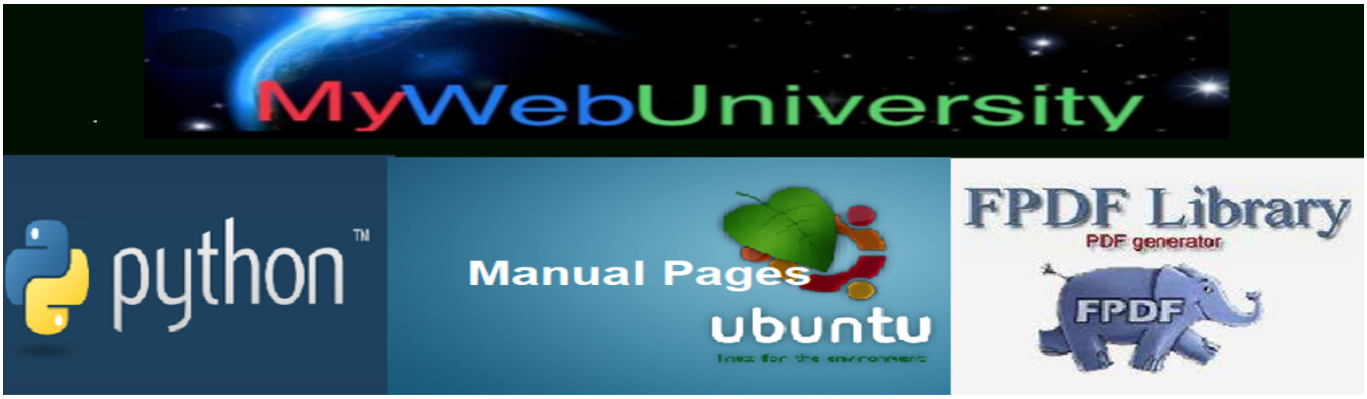

Rocky Enterprise Linux 9.2 Manual Pages on command 'lzcat.1'

$ man lzcat.1

XZ(1) XZ Utils XZ(1)

NAME

xz, unxz, xzcat, lzma, unlzma, lzcat - Compress or decompress .xz and .lzma files

SYNOPSIS

xz [option...] [file...]

COMMAND ALIASES

unxz is equivalent to xz --decompress.

xzcat is equivalent to xz --decompress --stdout.

lzma is equivalent to xz --format=lzma.

unlzma is equivalent to xz --format=lzma --decompress.

lzcat is equivalent to xz --format=lzma --decompress --stdout.

When writing scripts that need to decompress files, it is recommended to always use the name xz with appropriate arguments (xz -d or xz -dc) instead of the names unxz and xzcat.

DESCRIPTION

xz is a general-purpose data compression tool with command line syntax similar to gzip(1) and bzip2(1). The native file format is the .xz format, but the legacy .lzma format used by LZMA Utils and raw compressed streams with no container format headers are also sup? ported.

xz compresses or decompresses each file according to the selected operation mode. If no files are given or file is -, xz reads from standard input and writes the processed data to standard output. xz will refuse (display an error and skip the file) to write com? pressed data to standard output if it is a terminal. Similarly, xz will refuse to read compressed data from standard input if it is a terminal.

Unless --stdout is specified, files other than - are written to a new file whose name is derived from the source file name:

? When compressing, the suffix of the target file format (.xz or .lzma) is appended to the source filename to get the target filename.

? When decompressing, the .xz or .lzma suffix is removed from the filename to get the target filename. xz also recognizes the suffixes .txz and .tlz, and replaces them with the .tar suffix.

If the target file already exists, an error is displayed and the file is skipped.

Unless writing to standard output, xz will display a warning and skip the file if any of the following applies:

? File is not a regular file. Symbolic links are not followed, and thus they are not considered to be regular files.

? File has more than one hard link.

? File has setuid, setgid, or sticky bit set.

? The operation mode is set to compress and the file already has a suffix of the target file format (.xz or .txz when compressing to the .xz format, and .lzma or .tlz when compressing to the .lzma format).

? The operation mode is set to decompress and the file doesn't have a suffix of any of the supported file formats (.xz, .txz, .lzma, or .tlz).

After successfully compressing or decompressing the file, xz copies the owner, group, per? missions, access time, and modification time from the source file to the target file. If copying the group fails, the permissions are modified so that the target file doesn't be? come accessible to users who didn't have permission to access the source file. xz doesn't support copying other metadata like access control lists or extended attributes yet.

Once the target file has been successfully closed, the source file is removed unless --keep was specified. The source file is never removed if the output is written to stan? dard output.

Sending SIGINFO or SIGUSR1 to the xz process makes it print progress information to stan? dard error. This has only limited use since when standard error is a terminal, using --verbose will display an automatically updating progress indicator.

Memory usage

The memory usage of xz varies from a few hundred kilobytes to several gigabytes depending on the compression settings. The settings used when compressing a file determine the mem?

ory requirements of the decompressor. Typically the decompressor needs 5 % to 20 % of the amount of memory that the compressor needed when creating the file. For example, decompressing a file created with xz -9 currently requires 65 MiB of memory. Still, it is possible to have .xz files that require several gigabytes of memory to decompress.

Especially users of older systems may find the possibility of very large memory usage annoying. To prevent uncomfortable surprises, xz has a built-in memory usage limiter, which is disabled by default. While some operating systems provide ways to limit the memory usage of processes, relying on it wasn't deemed to be flexible enough (for example, using ulimit(1) to limit virtual memory tends to cripple mmap(2)).

The memory usage limiter can be enabled with the command line option --memlimit=limit. Often it is more convenient to enable the limiter by default by setting the environment variable XZ_DEFAULTS, for example, XZ_DEFAULTS=--memlimit=150MiB. It is possible to set the limits separately for compression and decompression by using --memlimit-compress=limit and --memlimit-decompress=limit. Using these two options outside XZ_DEFAULTS is rarely useful because a single run of xz cannot do both compression and decompression and --memlimit=limit (or -M limit) is shorter to type on the command line.

If the specified memory usage limit is exceeded when decompressing, xz will display an error and decompressing the file will fail. If the limit is exceeded when compressing, xz will try to scale the settings down so that the limit is no longer exceeded (except when using --format=raw or --no-adjust). This way the operation won't fail unless the limit is very small. The scaling of the settings is done in steps that don't match the compression level presets, for example, if the limit is only slightly less than the amount required for xz -9, the settings will be scaled down only a little, not all the way down to xz -8.

Concatenation and padding with .xz files

It is possible to concatenate .xz files as is. xz will decompress such files as if they were a single .xz file.

It is possible to insert padding between the concatenated parts or after the last part. The padding must consist of null bytes and the size of the padding must be a multiple of four bytes. This can be useful, for example, if the .xz file is stored on a medium that measures file sizes in 512-byte blocks.

Concatenation and padding are not allowed with .lzma files or raw streams.

OPTIONS

Integer suffixes and special values

In most places where an integer argument is expected, an optional suffix is supported to easily indicate large integers. There must be no space between the integer and the suf? fix.

KiB Multiply the integer by 1,024 (2^10). Ki, k, kB, K, and KB are accepted as syn? onyms for KiB.

MiB Multiply the integer by 1,048,576 (2^20). Mi, m, M, and MB are accepted as syn? onyms for MiB.

GiB Multiply the integer by 1,073,741,824 (2^30). Gi, g, G, and GB are accepted as synonyms for GiB.

The special value max can be used to indicate the maximum integer value supported by the option.

Operation mode

If multiple operation mode options are given, the last one takes effect.

-z, --compress

Compress. This is the default operation mode when no operation mode option is specified and no other operation mode is implied from the command name (for exam? ple, unxz implies --decompress).

-d, --decompress, --uncompress

Decompress.

-t, --test

Test the integrity of compressed files. This option is equivalent to --decompress --stdout except that the decompressed data is discarded instead of being written to standard output. No files are created or removed.

-l, --list

Print information about compressed files. No uncompressed output is produced, and no files are created or removed. In list mode, the program cannot read the com? pressed data from standard input or from other unseekable sources.

The default listing shows basic information about files, one file per line. To get more detailed information, use also the --verbose option. For even more informa? tion, use --verbose twice, but note that this may be slow, because getting all the extra information requires many seeks. The width of verbose output exceeds 80 characters, so piping the output to, for example, less -S may be convenient if the terminal isn't wide enough.

The exact output may vary between xz versions and different locales. For machine-readable output, --robot --list should be used.

Operation modifiers

-k, --keep

Don't delete the input files.

Since xz 5.4.0, this option also makes xz compress or decompress even if the input is a symbolic link to a regular file, has more than one hard link, or has the se? tuid, setgid, or sticky bit set. The setuid, setgid, and sticky bits are not copied to the target file. In earlier versions this was only done with --force.

-f, --force

This option has several effects:

? If the target file already exists, delete it before compressing or decompress? ing.

? Compress or decompress even if the input is a symbolic link to a regular file, has more than one hard link, or has the setuid, setgid, or sticky bit set. The setuid, setgid, and sticky bits are not copied to the target file.

? When used with --decompress --stdout and xz cannot recognize the type of the source file, copy the source file as is to standard output. This allows xzcat --force to be used like cat(1) for files that have not been compressed with xz. Note that in future, xz might support new compressed file formats, which may make xz decompress more types of files instead of copying them as is to standard output. --format=format can be used to restrict xz to decompress only a single file format.

-c, --stdout, --to-stdout

Write the compressed or decompressed data to standard output instead of a file. This implies --keep.

--single-stream

Decompress only the first .xz stream, and silently ignore possible remaining input data following the stream. Normally such trailing garbage makes xz display an er? ror.

xz never decompresses more than one stream from .lzma files or raw streams, but this option still makes xz ignore the possible trailing data after the .lzma file or raw stream.

This option has no effect if the operation mode is not --decompress or --test.

--no-sparse

Disable creation of sparse files. By default, if decompressing into a regular file, xz tries to make the file sparse if the decompressed data contains long se? quences of binary zeros. It also works when writing to standard output as long as standard output is connected to a regular file and certain additional conditions are met to make it safe. Creating sparse files may save disk space and speed up the decompression by reducing the amount of disk I/O.

-S .suf, --suffix=.suf

When compressing, use .suf as the suffix for the target file instead of .xz or .lzma. If not writing to standard output and the source file already has the suf? fix .suf, a warning is displayed and the file is skipped.

When decompressing, recognize files with the suffix .suf in addition to files with the .xz, .txz, .lzma, or .tlz suffix. If the source file has the suffix .suf, the suffix is removed to get the target filename.

When compressing or decompressing raw streams (--format=raw), the suffix must al? ways be specified unless writing to standard output, because there is no default suffix for raw streams.

--files[=file]

Read the filenames to process from file; if file is omitted, filenames are read from standard input. Filenames must be terminated with the newline character. A dash (-) is taken as a regular filename; it doesn't mean standard input. If file? names are given also as command line arguments, they are processed before the file? names read from file.

--files0[=file]

This is identical to --files[=file] except that each filename must be terminated with the null character.

Basic file format and compression options

-F format, --format=format

Specify the file format to compress or decompress:

auto This is the default. When compressing, auto is equivalent to xz. When de? compressing, the format of the input file is automatically detected. Note that raw streams (created with --format=raw) cannot be auto-detected.

xz Compress to the .xz file format, or accept only .xz files when decompress?
ing.

lzma, alone

Compress to the legacy .lzma file format, or accept only .lzma files when decompressing. The alternative name alone is provided for backwards compat?
ibility with LZMA Utils.

raw Compress or uncompress a raw stream (no headers). This is meant for ad?
vanced users only. To decode raw streams, you need use --format=raw and ex?
plicitly specify the filter chain, which normally would have been stored in
the container headers.

-C check, --check=check

Specify the type of the integrity check. The check is calculated from the uncom?
pressed data and stored in the .xz file. This option has an effect only when com?
pressing into the .xz format; the .lzma format doesn't support integrity checks.
The integrity check (if any) is verified when the .xz file is decompressed.

Supported check types:

none Don't calculate an integrity check at all. This is usually a bad idea.
This can be useful when integrity of the data is verified by other means
anyway.

crc32 Calculate CRC32 using the polynomial from IEEE-802.3 (Ethernet).

crc64 Calculate CRC64 using the polynomial from ECMA-182. This is the default,
since it is slightly better than CRC32 at detecting damaged files and the
speed difference is negligible.

sha256 Calculate SHA-256. This is somewhat slower than CRC32 and CRC64.

Integrity of the .xz headers is always verified with CRC32. It is not possible to
change or disable it.

--ignore-check

Don't verify the integrity check of the compressed data when decompressing. The
CRC32 values in the .xz headers will still be verified normally.

Do not use this option unless you know what you are doing. Possible reasons to use
this option:

? Trying to recover data from a corrupt .xz file.

? Speeding up decompression. This matters mostly with SHA-256 or with files that

have compressed extremely well. It's recommended to not use this option for this purpose unless the file integrity is verified externally in some other way.

-0 ... -9

Select a compression preset level. The default is -6. If multiple preset levels are specified, the last one takes effect. If a custom filter chain was already specified, setting a compression preset level clears the custom filter chain.

The differences between the presets are more significant than with gzip(1) and bzip2(1). The selected compression settings determine the memory requirements of the decompressor, thus using a too high preset level might make it painful to de? compress the file on an old system with little RAM. Specifically, it's not a good idea to blindly use -9 for everything like it often is with gzip(1) and bzip2(1).

-0 ... -3

These are somewhat fast presets. -0 is sometimes faster than gzip -9 while compressing much better. The higher ones often have speed comparable to bzip2(1) with comparable or better compression ratio, although the results depend a lot on the type of data being compressed.

-4 ... -6

Good to very good compression while keeping decompressor memory usage rea? sonable even for old systems. -6 is the default, which is usually a good choice for distributing files that need to be decompressible even on systems with only 16 MiB RAM. (-5e or -6e may be worth considering too. See --ex? treme.)

-7 ... -9

These are like -6 but with higher compressor and decompressor memory re? quirements. These are useful only when compressing files bigger than 8 MiB, 16 MiB, and 32 MiB, respectively.

On the same hardware, the decompression speed is approximately a constant number of bytes of compressed data per second. In other words, the better the compression, the faster the decompression will usually be. This also means that the amount of uncompressed output produced per second can vary a lot.

The following table summarises the features of the presets:

Preset	DictSize	CompCPU	CompMem	DecMem
-0	256 KiB	0	3 MiB	1 MiB

-1	1 MiB	1	9 MiB	2 MiB
-2	2 MiB	2	17 MiB	3 MiB
-3	4 MiB	3	32 MiB	5 MiB
-4	4 MiB	4	48 MiB	5 MiB
-5	8 MiB	5	94 MiB	9 MiB
-6	8 MiB	6	94 MiB	9 MiB
-7	16 MiB	6	186 MiB	17 MiB
-8	32 MiB	6	370 MiB	33 MiB
-9	64 MiB	6	674 MiB	65 MiB

Column descriptions:

? DictSize is the LZMA2 dictionary size. It is waste of memory to use a dictio?
nary bigger than the size of the uncompressed file. This is why it is good to
avoid using the presets -7 ... -9 when there's no real need for them. At -6 and
lower, the amount of memory wasted is usually low enough to not matter.

? CompCPU is a simplified representation of the LZMA2 settings that affect com?
pression speed. The dictionary size affects speed too, so while CompCPU is the
same for levels -6 ... -9, higher levels still tend to be a little slower. To
get even slower and thus possibly better compression, see --extreme.

? CompMem contains the compressor memory requirements in the single-threaded mode.
It may vary slightly between xz versions. Memory requirements of some of the
future multithreaded modes may be dramatically higher than that of the single-
threaded mode.

? DecMem contains the decompressor memory requirements. That is, the compression
settings determine the memory requirements of the decompressor. The exact de?
compressor memory usage is slightly more than the LZMA2 dictionary size, but the
values in the table have been rounded up to the next full MiB.

-e, --extreme

Use a slower variant of the selected compression preset level (-0 ... -9) to hope?
fully get a little bit better compression ratio, but with bad luck this can also
make it worse. Decompressor memory usage is not affected, but compressor memory
usage increases a little at preset levels -0 ... -3.

Since there are two presets with dictionary sizes 4 MiB and 8 MiB, the presets -3e
and -5e use slightly faster settings (lower CompCPU) than -4e and -6e, respec?

tively. That way no two presets are identical.

Preset	DictSize	CompCPU	CompMem	DecMem
-0e	256 KiB	8	4 MiB	1 MiB
-1e	1 MiB	8	13 MiB	2 MiB
-2e	2 MiB	8	25 MiB	3 MiB
-3e	4 MiB	7	48 MiB	5 MiB
-4e	4 MiB	8	48 MiB	5 MiB
-5e	8 MiB	7	94 MiB	9 MiB
-6e	8 MiB	8	94 MiB	9 MiB
-7e	16 MiB	8	186 MiB	17 MiB
-8e	32 MiB	8	370 MiB	33 MiB
-9e	64 MiB	8	674 MiB	65 MiB

For example, there are a total of four presets that use 8 MiB dictionary, whose or? der from the fastest to the slowest is -5, -6, -5e, and -6e.

--fast

--best These are somewhat misleading aliases for -0 and -9, respectively. These are pro? vided only for backwards compatibility with LZMA Utils. Avoid using these options.

--block-size=size

When compressing to the .xz format, split the input data into blocks of size bytes. The blocks are compressed independently from each other, which helps with multi- threading and makes limited random-access decompression possible. This option is typically used to override the default block size in multi-threaded mode, but this option can be used in single-threaded mode too.

In multi-threaded mode about three times size bytes will be allocated in each thread for buffering input and output. The default size is three times the LZMA2 dictionary size or 1 MiB, whichever is more. Typically a good value is 2-4 times the size of the LZMA2 dictionary or at least 1 MiB. Using size less than the LZMA2 dictionary size is waste of RAM because then the LZMA2 dictionary buffer will never get fully used. The sizes of the blocks are stored in the block headers, which a future version of xz will use for multi-threaded decompression.

In single-threaded mode no block splitting is done by default. Setting this option doesn't affect memory usage. No size information is stored in block headers, thus files created in single-threaded mode won't be identical to files created in multi-

threaded mode. The lack of size information also means that a future version of xz won't be able decompress the files in multi-threaded mode.

--block-list=sizes

When compressing to the .xz format, start a new block after the given intervals of uncompressed data.

The uncompressed sizes of the blocks are specified as a comma-separated list. Omitting a size (two or more consecutive commas) is a shorthand to use the size of the previous block.

If the input file is bigger than the sum of sizes, the last value in sizes is re? peated until the end of the file. A special value of 0 may be used as the last value to indicate that the rest of the file should be encoded as a single block.

If one specifies sizes that exceed the encoder's block size (either the default value in threaded mode or the value specified with --block-size=size), the encoder will create additional blocks while keeping the boundaries specified in sizes. For example, if one specifies --block-size=10MiB --block-list=5MiB,10MiB,8MiB,12MiB,24MiB and the input file is 80 MiB, one will get 11 blocks: 5, 10, 8, 10, 2, 10, 10, 4, 10, 10, and 1 MiB.

In multi-threaded mode the sizes of the blocks are stored in the block headers. This isn't done in single-threaded mode, so the encoded output won't be identical to that of the multi-threaded mode.

--flush-timeout=timeout

When compressing, if more than timeout milliseconds (a positive integer) has passed since the previous flush and reading more input would block, all the pending input data is flushed from the encoder and made available in the output stream. This can be useful if xz is used to compress data that is streamed over a network. Small timeout values make the data available at the receiving end with a small delay, but large timeout values give better compression ratio.

This feature is disabled by default. If this option is specified more than once, the last one takes effect. The special timeout value of 0 can be used to explic? itly disable this feature.

This feature is not available on non-POSIX systems.

This feature is still experimental. Currently xz is unsuitable for decompressing the stream in real time due to how xz does buffering.

--memlimit-compress=limit

Set a memory usage limit for compression. If this option is specified multiple times, the last one takes effect.

If the compression settings exceed the limit, xz will adjust the settings downwards so that the limit is no longer exceeded and display a notice that automatic adjustment was done. Such adjustments are not made when compressing with --format=raw or if --no-adjust has been specified. In those cases, an error is displayed and xz will exit with exit status 1.

The limit can be specified in multiple ways:

? The limit can be an absolute value in bytes. Using an integer suffix like MiB can be useful. Example: --memlimit-compress=80MiB

? The limit can be specified as a percentage of total physical memory (RAM). This can be useful especially when setting the XZ_DEFAULTS environment variable in a shell initialization script that is shared between different computers. That way the limit is automatically bigger on systems with more memory. Example: --memlimit-compress=70%

? The limit can be reset back to its default value by setting it to 0. This is currently equivalent to setting the limit to max (no memory usage limit). Once multithreading support has been implemented, there may be a difference between 0 and max for the multithreaded case, so it is recommended to use 0 instead of max until the details have been decided.

For 32-bit xz there is a special case: if the limit would be over 4020 MiB, the limit is set to 4020 MiB. (The values 0 and max aren't affected by this. A similar feature doesn't exist for decompression.) This can be helpful when a 32-bit executable has access to 4 GiB address space while hopefully doing no harm in other situations.

See also the section Memory usage.

--memlimit-decompress=limit

Set a memory usage limit for decompression. This also affects the --list mode. If the operation is not possible without exceeding the limit, xz will display an error and decompressing the file will fail. See --memlimit-compress=limit for possible ways to specify the limit.

-M limit, --memlimit=limit, --memory=limit

This is equivalent to specifying --memlimit-compress=limit --memlimit-decom? press=limit.

--no-adjust

Display an error and exit if the compression settings exceed the memory usage limit. The default is to adjust the settings downwards so that the memory usage limit is not exceeded. Automatic adjusting is always disabled when creating raw streams (--format=raw).

-T threads, --threads=threads

Specify the number of worker threads to use. Setting threads to a special value 0 makes xz use as many threads as there are CPU cores on the system. The actual num? ber of threads can be less than threads if the input file is not big enough for threading with the given settings or if using more threads would exceed the memory usage limit.

Currently the only threading method is to split the input into blocks and compress them independently from each other. The default block size depends on the compres? sion level and can be overridden with the --block-size=size option.

Threaded decompression hasn't been implemented yet. It will only work on files that contain multiple blocks with size information in block headers. All files compressed in multi-threaded mode meet this condition, but files compressed in sin? gle-threaded mode don't even if --block-size=size is used.

Custom compressor filter chains

A custom filter chain allows specifying the compression settings in detail instead of re? lying on the settings associated to the presets. When a custom filter chain is specified, preset options (-0 ... -9 and --extreme) earlier on the command line are forgotten. If a preset option is specified after one or more custom filter chain options, the new preset takes effect and the custom filter chain options specified earlier are forgotten.

A filter chain is comparable to piping on the command line. When compressing, the uncom? pressed input goes to the first filter, whose output goes to the next filter (if any). The output of the last filter gets written to the compressed file. The maximum number of filters in the chain is four, but typically a filter chain has only one or two filters.

Many filters have limitations on where they can be in the filter chain: some filters can work only as the last filter in the chain, some only as a non-last filter, and some work in any position in the chain. Depending on the filter, this limitation is either inherent

to the filter design or exists to prevent security issues.

A custom filter chain is specified by using one or more filter options in the order they are wanted in the filter chain. That is, the order of filter options is significant!

When decoding raw streams (--format=raw), the filter chain is specified in the same order as it was specified when compressing.

Filters take filter-specific options as a comma-separated list. Extra commas in options are ignored. Every option has a default value, so you need to specify only those you want to change.

To see the whole filter chain and options, use xz -vv (that is, use --verbose twice). This works also for viewing the filter chain options used by presets.

--lzma1[=options]

--lzma2[=options]

Add LZMA1 or LZMA2 filter to the filter chain. These filters can be used only as the last filter in the chain.

LZMA1 is a legacy filter, which is supported almost solely due to the legacy .lzma file format, which supports only LZMA1. LZMA2 is an updated version of LZMA1 to fix some practical issues of LZMA1. The .xz format uses LZMA2 and doesn't support LZMA1 at all. Compression speed and ratios of LZMA1 and LZMA2 are practically the same.

LZMA1 and LZMA2 share the same set of options:

preset=preset

Reset all LZMA1 or LZMA2 options to preset. Preset consist of an integer, which may be followed by single-letter preset modifiers. The integer can be from 0 to 9, matching the command line options -0 ... -9. The only sup? ported modifier is currently e, which matches --extreme. If no preset is specified, the default values of LZMA1 or LZMA2 options are taken from the preset 6.

dict=size

Dictionary (history buffer) size indicates how many bytes of the recently processed uncompressed data is kept in memory. The algorithm tries to find repeating byte sequences (matches) in the uncompressed data, and replace them with references to the data currently in the dictionary. The bigger the dictionary, the higher is the chance to find a match. Thus, increasing

dictionary size usually improves compression ratio, but a dictionary bigger than the uncompressed file is waste of memory.

Typical dictionary size is from 64 KiB to 64 MiB. The minimum is 4 KiB. The maximum for compression is currently 1.5 GiB (1536 MiB). The decompressor already supports dictionaries up to one byte less than 4 GiB, which is the maximum for the LZMA1 and LZMA2 stream formats.

Dictionary size and match finder (mf) together determine the memory usage of the LZMA1 or LZMA2 encoder. The same (or bigger) dictionary size is required for decompressing that was used when compressing, thus the memory usage of the decoder is determined by the dictionary size used when compressing. The .xz headers store the dictionary size either as 2^n or 2^n + 2^(n-1), so these sizes are somewhat preferred for compression. Other sizes will get rounded up when stored in the .xz headers.

lc=lc Specify the number of literal context bits. The minimum is 0 and the maximum is 4; the default is 3. In addition, the sum of lc and lp must not exceed 4.

All bytes that cannot be encoded as matches are encoded as literals. That is, literals are simply 8-bit bytes that are encoded one at a time.

The literal coding makes an assumption that the highest lc bits of the previous uncompressed byte correlate with the next byte. For example, in typical English text, an upper-case letter is often followed by a lower-case letter, and a lower-case letter is usually followed by another lower-case letter. In the US-ASCII character set, the highest three bits are 010 for upper-case letters and 011 for lower-case letters. When lc is at least 3, the literal coding can take advantage of this property in the uncompressed data.

The default value (3) is usually good. If you want maximum compression, test lc=4. Sometimes it helps a little, and sometimes it makes compression worse. If it makes it worse, test lc=2 too.

lp=lp Specify the number of literal position bits. The minimum is 0 and the maximum is 4; the default is 0.

Lp affects what kind of alignment in the uncompressed data is assumed when encoding literals. See pb below for more information about alignment.

pb=pb Specify the number of position bits. The minimum is 0 and the maximum is 4;

the default is 2.

Pb affects what kind of alignment in the uncompressed data is assumed in general. The default means four-byte alignment (2^pb=2^2=4), which is often a good choice when there's no better guess.

When the aligment is known, setting pb accordingly may reduce the file size a little. For example, with text files having one-byte alignment (US-ASCII, ISO-8859-*, UTF-8), setting pb=0 can improve compression slightly. For UTF-16 text, pb=1 is a good choice. If the alignment is an odd number like 3 bytes, pb=0 might be the best choice.

Even though the assumed alignment can be adjusted with pb and lp, LZMA1 and LZMA2 still slightly favor 16-byte alignment. It might be worth taking into account when designing file formats that are likely to be often compressed with LZMA1 or LZMA2.

mf=mf Match finder has a major effect on encoder speed, memory usage, and compres? sion ratio. Usually Hash Chain match finders are faster than Binary Tree match finders. The default depends on the preset: 0 uses hc3, 1-3 use hc4, and the rest use bt4.

The following match finders are supported. The memory usage formulas below are rough approximations, which are closest to the reality when dict is a power of two.

hc3 Hash Chain with 2- and 3-byte hashing

Minimum value for nice: 3

Memory usage:

dict * 7.5 (if dict <= 16 MiB);

dict * 5.5 + 64 MiB (if dict > 16 MiB)

hc4 Hash Chain with 2-, 3-, and 4-byte hashing

Minimum value for nice: 4

Memory usage:

dict * 7.5 (if dict <= 32 MiB);

dict * 6.5 (if dict > 32 MiB)

bt2 Binary Tree with 2-byte hashing

Minimum value for nice: 2

Memory usage: dict * 9.5

bt3 Binary Tree with 2- and 3-byte hashing

Minimum value for nice: 3

Memory usage:

dict * 11.5 (if dict <= 16 MiB);

dict * 9.5 + 64 MiB (if dict > 16 MiB)

bt4 Binary Tree with 2-, 3-, and 4-byte hashing

Minimum value for nice: 4

Memory usage:

dict * 11.5 (if dict <= 32 MiB);

dict * 10.5 (if dict > 32 MiB)

mode=mode

Compression mode specifies the method to analyze the data produced by the match finder. Supported modes are fast and normal. The default is fast for presets 0-3 and normal for presets 4-9.

Usually fast is used with Hash Chain match finders and normal with Binary Tree match finders. This is also what the presets do.

nice=nice

Specify what is considered to be a nice length for a match. Once a match of at least nice bytes is found, the algorithm stops looking for possibly bet? ter matches.

Nice can be 2-273 bytes. Higher values tend to give better compression ra? tio at the expense of speed. The default depends on the preset.

depth=depth

Specify the maximum search depth in the match finder. The default is the special value of 0, which makes the compressor determine a reasonable depth from mf and nice.

Reasonable depth for Hash Chains is 4-100 and 16-1000 for Binary Trees. Us? ing very high values for depth can make the encoder extremely slow with some files. Avoid setting the depth over 1000 unless you are prepared to inter? rupt the compression in case it is taking far too long.

When decoding raw streams (--format=raw), LZMA2 needs only the dictionary size. LZMA1 needs also lc, lp, and pb.

--x86[=options]

--powerpc[=options]

--ia64[=options]

--arm[=options]

--armthumb[=options]

--sparc[=options]

Add a branch/call/jump (BCJ) filter to the filter chain. These filters can be used only as a non-last filter in the filter chain.

A BCJ filter converts relative addresses in the machine code to their absolute counterparts. This doesn't change the size of the data, but it increases redun? dancy, which can help LZMA2 to produce 0-15 % smaller .xz file. The BCJ filters are always reversible, so using a BCJ filter for wrong type of data doesn't cause any data loss, although it may make the compression ratio slightly worse.

It is fine to apply a BCJ filter on a whole executable; there's no need to apply it only on the executable section. Applying a BCJ filter on an archive that contains both executable and non-executable files may or may not give good results, so it generally isn't good to blindly apply a BCJ filter when compressing binary packages for distribution.

These BCJ filters are very fast and use insignificant amount of memory. If a BCJ filter improves compression ratio of a file, it can improve decompression speed at the same time. This is because, on the same hardware, the decompression speed of LZMA2 is roughly a fixed number of bytes of compressed data per second.

These BCJ filters have known problems related to the compression ratio:

? Some types of files containing executable code (for example, object files, static libraries, and Linux kernel modules) have the addresses in the instruc? tions filled with filler values. These BCJ filters will still do the address conversion, which will make the compression worse with these files.

? Applying a BCJ filter on an archive containing multiple similar executables can make the compression ratio worse than not using a BCJ filter. This is because the BCJ filter doesn't detect the boundaries of the executable files, and doesn't reset the address conversion counter for each executable.

Both of the above problems will be fixed in the future in a new filter. The old BCJ filters will still be useful in embedded systems, because the decoder of the

new filter will be bigger and use more memory.

Different instruction sets have different alignment:

Filter	Alignment	Notes
x86	1	32-bit or 64-bit x86
PowerPC	4	Big endian only
ARM	4	Little endian only
ARM-Thumb	2	Little endian only
IA-64	16	Big or little endian
SPARC	4	Big or little endian

Since the BCJ-filtered data is usually compressed with LZMA2, the compression ratio may be improved slightly if the LZMA2 options are set to match the alignment of the selected BCJ filter. For example, with the IA-64 filter, it's good to set pb=4 with LZMA2 (2^4=16). The x86 filter is an exception; it's usually good to stick to LZMA2's default four-byte alignment when compressing x86 executables.

All BCJ filters support the same options:

start=offset

Specify the start offset that is used when converting between relative and absolute addresses. The offset must be a multiple of the alignment of the filter (see the table above). The default is zero. In practice, the de? fault is good; specifying a custom offset is almost never useful.

--delta[=options]

Add the Delta filter to the filter chain. The Delta filter can be only used as a non-last filter in the filter chain.

Currently only simple byte-wise delta calculation is supported. It can be useful when compressing, for example, uncompressed bitmap images or uncompressed PCM au? dio. However, special purpose algorithms may give significantly better results than Delta + LZMA2. This is true especially with audio, which compresses faster and better, for example, with flac(1).

Supported options:

dist=distance

Specify the distance of the delta calculation in bytes. distance must be 1-256. The default is 1.

For example, with dist=2 and eight-byte input A1 B1 A2 B3 A3 B5 A4 B7, the

output will be A1 B1 01 02 01 02 01 02.

Other options

-q, --quiet

Suppress warnings and notices. Specify this twice to suppress errors too. This option has no effect on the exit status. That is, even if a warning was sup? pressed, the exit status to indicate a warning is still used.

-v, --verbose

Be verbose. If standard error is connected to a terminal, xz will display a progress indicator. Specifying --verbose twice will give even more verbose output. The progress indicator shows the following information:

? Completion percentage is shown if the size of the input file is known. That is, the percentage cannot be shown in pipes.

? Amount of compressed data produced (compressing) or consumed (decompressing).

? Amount of uncompressed data consumed (compressing) or produced (decompressing).

? Compression ratio, which is calculated by dividing the amount of compressed data processed so far by the amount of uncompressed data processed so far.

? Compression or decompression speed. This is measured as the amount of uncom? pressed data consumed (compression) or produced (decompression) per second. It is shown after a few seconds have passed since xz started processing the file.

? Elapsed time in the format M:SS or H:MM:SS.

? Estimated remaining time is shown only when the size of the input file is known and a couple of seconds have already passed since xz started processing the file. The time is shown in a less precise format which never has any colons, for example, 2 min 30 s.

When standard error is not a terminal, --verbose will make xz print the filename, compressed size, uncompressed size, compression ratio, and possibly also the speed and elapsed time on a single line to standard error after compressing or decom? pressing the file. The speed and elapsed time are included only when the operation took at least a few seconds. If the operation didn't finish, for example, due to user interruption, also the completion percentage is printed if the size of the in? put file is known.

-Q, --no-warn

Don't set the exit status to 2 even if a condition worth a warning was detected.

This option doesn't affect the verbosity level, thus both --quiet and --no-warn have to be used to not display warnings and to not alter the exit status.

--robot

Print messages in a machine-parsable format. This is intended to ease writing frontends that want to use xz instead of liblzma, which may be the case with vari? ous scripts. The output with this option enabled is meant to be stable across xz releases. See the section ROBOT MODE for details.

--info-memory

Display, in human-readable format, how much physical memory (RAM) xz thinks the system has and the memory usage limits for compression and decompression, and exit successfully.

-h, --help

Display a help message describing the most commonly used options, and exit success? fully.

-H, --long-help

Display a help message describing all features of xz, and exit successfully

-V, --version

Display the version number of xz and liblzma in human readable format. To get ma? chine-parsable output, specify --robot before --version.

ROBOT MODE

The robot mode is activated with the --robot option. It makes the output of xz easier to parse by other programs. Currently --robot is supported only together with --version, --info-memory, and --list. It will be supported for compression and decompression in the future.

Version

xz --robot --version will print the version number of xz and liblzma in the following for? mat:

XZ_VERSION=XYYYZZZS

LIBLZMA_VERSION=XYYYZZZS

X Major version.

YYY Minor version. Even numbers are stable. Odd numbers are alpha or beta versions.

ZZZ Patch level for stable releases or just a counter for development releases.

S Stability. 0 is alpha, 1 is beta, and 2 is stable. S should be always 2 when YYY

is even.

XYYYZZZS are the same on both lines if xz and liblzma are from the same XZ Utils release.

Examples: 4.999.9beta is 49990091 and 5.0.0 is 50000002.

Memory limit information

xz --robot --info-memory prints a single line with three tab-separated columns:

1. Total amount of physical memory (RAM) in bytes
2. Memory usage limit for compression in bytes. A special value of zero indicates the default setting, which for single-threaded mode is the same as no limit.
3. Memory usage limit for decompression in bytes. A special value of zero indicates the default setting, which for single-threaded mode is the same as no limit.

In the future, the output of xz --robot --info-memory may have more columns, but never more than a single line.

List mode

xz --robot --list uses tab-separated output. The first column of every line has a string that indicates the type of the information found on that line:

name This is always the first line when starting to list a file. The second column on the line is the filename.

file This line contains overall information about the .xz file. This line is always printed after the name line.

stream This line type is used only when --verbose was specified. There are as many stream lines as there are streams in the .xz file.

block This line type is used only when --verbose was specified. There are as many block lines as there are blocks in the .xz file. The block lines are shown after all the stream lines; different line types are not interleaved.

summary

This line type is used only when --verbose was specified twice. This line is printed after all block lines. Like the file line, the summary line contains over? all information about the .xz file.

totals This line is always the very last line of the list output. It shows the total counts and sizes.

The columns of the file lines:

2. Number of streams in the file
3. Total number of blocks in the stream(s)

4. Compressed size of the file

5. Uncompressed size of the file

6. Compression ratio, for example, 0.123. If ratio is over 9.999, three dashes (---) are displayed instead of the ratio.

7. Comma-separated list of integrity check names. The following strings are used for the known check types: None, CRC32, CRC64, and SHA-256. For unknown check types, Unknown-N is used, where N is the Check ID as a decimal number (one or two digits).

8. Total size of stream padding in the file

The columns of the stream lines:

2. Stream number (the first stream is 1)

3. Number of blocks in the stream

4. Compressed start offset

5. Uncompressed start offset

6. Compressed size (does not include stream padding)

7. Uncompressed size

8. Compression ratio

9. Name of the integrity check

10. Size of stream padding

The columns of the block lines:

2. Number of the stream containing this block

3. Block number relative to the beginning of the stream (the first block is 1)

4. Block number relative to the beginning of the file

5. Compressed start offset relative to the beginning of the file

6. Uncompressed start offset relative to the beginning of the file

7. Total compressed size of the block (includes headers)

8. Uncompressed size

9. Compression ratio

10. Name of the integrity check

If --verbose was specified twice, additional columns are included on the block lines. These are not displayed with a single --verbose, because getting this information requires many seeks and can thus be slow:

11. Value of the integrity check in hexadecimal

12. Block header size

13. Block flags: c indicates that compressed size is present, and u indicates that uncompressed size is present. If the flag is not set, a dash (-) is shown in? stead to keep the string length fixed. New flags may be added to the end of the string in the future.

14. Size of the actual compressed data in the block (this excludes the block header, block padding, and check fields)

15. Amount of memory (in bytes) required to decompress this block with this xz ver? sion

16. Filter chain. Note that most of the options used at compression time cannot be known, because only the options that are needed for decompression are stored in the .xz headers.

The columns of the summary lines:

2. Amount of memory (in bytes) required to decompress this file with this xz ver? sion

3. yes or no indicating if all block headers have both compressed size and uncom? pressed size stored in them

Since xz 5.1.2alpha:

4. Minimum xz version required to decompress the file

The columns of the totals line:

2. Number of streams

3. Number of blocks

4. Compressed size

5. Uncompressed size

6. Average compression ratio

7. Comma-separated list of integrity check names that were present in the files

8. Stream padding size

9. Number of files. This is here to keep the order of the earlier columns the same as on file lines.

If --verbose was specified twice, additional columns are included on the totals line:

10. Maximum amount of memory (in bytes) required to decompress the files with this xz version

11. yes or no indicating if all block headers have both compressed size and uncom?

pressed size stored in them

Since xz 5.1.2alpha:

12. Minimum xz version required to decompress the file

Future versions may add new line types and new columns can be added to the existing line types, but the existing columns won't be changed.

EXIT STATUS

0 All is good.

1 An error occurred.

2 Something worth a warning occurred, but no actual errors occurred.

Notices (not warnings or errors) printed on standard error don't affect the exit status.

ENVIRONMENT

xz parses space-separated lists of options from the environment variables XZ_DEFAULTS and XZ_OPT, in this order, before parsing the options from the command line. Note that only options are parsed from the environment variables; all non-options are silently ignored. Parsing is done with getopt_long(3) which is used also for the command line arguments.

XZ_DEFAULTS

User-specific or system-wide default options. Typically this is set in a shell initialization script to enable xz's memory usage limiter by default. Excluding shell initialization scripts and similar special cases, scripts must never set or unset XZ_DEFAULTS.

XZ_OPT This is for passing options to xz when it is not possible to set the options di? rectly on the xz command line. This is the case when xz is run by a script or tool, for example, GNU tar(1):

XZ_OPT=-2v tar caf foo.tar.xz foo

Scripts may use XZ_OPT, for example, to set script-specific default compression op? tions. It is still recommended to allow users to override XZ_OPT if that is rea? sonable. For example, in sh(1) scripts one may use something like this:

XZ_OPT=${XZ_OPT-"-7e"}

export XZ_OPT

LZMA UTILS COMPATIBILITY

The command line syntax of xz is practically a superset of lzma, unlzma, and lzcat as found from LZMA Utils 4.32.x. In most cases, it is possible to replace LZMA Utils with XZ Utils without breaking existing scripts. There are some incompatibilities though, which

may sometimes cause problems.

Compression preset levels

The numbering of the compression level presets is not identical in xz and LZMA Utils. The most important difference is how dictionary sizes are mapped to different presets. Dic? tionary size is roughly equal to the decompressor memory usage.

Level	xz	LZMA Utils
-0	256 KiB	N/A
-1	1 MiB	64 KiB
-2	2 MiB	1 MiB
-3	4 MiB	512 KiB
-4	4 MiB	1 MiB
-5	8 MiB	2 MiB
-6	8 MiB	4 MiB
-7	16 MiB	8 MiB
-8	32 MiB	16 MiB
-9	64 MiB	32 MiB

The dictionary size differences affect the compressor memory usage too, but there are some other differences between LZMA Utils and XZ Utils, which make the difference even bigger:

Level	xz	LZMA Utils 4.32.x
-0	3 MiB	N/A
-1	9 MiB	2 MiB
-2	17 MiB	12 MiB
-3	32 MiB	12 MiB
-4	48 MiB	16 MiB
-5	94 MiB	26 MiB
-6	94 MiB	45 MiB
-7	186 MiB	83 MiB
-8	370 MiB	159 MiB
-9	674 MiB	311 MiB

The default preset level in LZMA Utils is -7 while in XZ Utils it is -6, so both use an 8 MiB dictionary by default.

Streamed vs. non-streamed .lzma files

The uncompressed size of the file can be stored in the .lzma header. LZMA Utils does that

when compressing regular files. The alternative is to mark that uncompressed size is un? known and use end-of-payload marker to indicate where the decompressor should stop. LZMA Utils uses this method when uncompressed size isn't known, which is the case, for example, in pipes.

xz supports decompressing .lzma files with or without end-of-payload marker, but all .lzma files created by xz will use end-of-payload marker and have uncompressed size marked as unknown in the .lzma header. This may be a problem in some uncommon situations. For ex? ample, a .lzma decompressor in an embedded device might work only with files that have known uncompressed size. If you hit this problem, you need to use LZMA Utils or LZMA SDK to create .lzma files with known uncompressed size.

Unsupported .lzma files

The .lzma format allows lc values up to 8, and lp values up to 4. LZMA Utils can decom? press files with any lc and lp, but always creates files with lc=3 and lp=0. Creating files with other lc and lp is possible with xz and with LZMA SDK.

The implementation of the LZMA1 filter in liblzma requires that the sum of lc and lp must not exceed 4. Thus, .lzma files, which exceed this limitation, cannot be decompressed with xz.

LZMA Utils creates only .lzma files which have a dictionary size of 2^n (a power of 2) but accepts files with any dictionary size. liblzma accepts only .lzma files which have a dictionary size of 2^n or 2^n + 2^(n-1). This is to decrease false positives when detect? ing .lzma files.

These limitations shouldn't be a problem in practice, since practically all .lzma files have been compressed with settings that liblzma will accept.

Trailing garbage

When decompressing, LZMA Utils silently ignore everything after the first .lzma stream. In most situations, this is a bug. This also means that LZMA Utils don't support decom? pressing concatenated .lzma files.

If there is data left after the first .lzma stream, xz considers the file to be corrupt unless --single-stream was used. This may break obscure scripts which have assumed that trailing garbage is ignored.

NOTES

Compressed output may vary

The exact compressed output produced from the same uncompressed input file may vary be?

tween XZ Utils versions even if compression options are identical. This is because the encoder can be improved (faster or better compression) without affecting the file format. The output can vary even between different builds of the same XZ Utils version, if differ? ent build options are used.

The above means that once --rsyncable has been implemented, the resulting files won't nec? essarily be rsyncable unless both old and new files have been compressed with the same xz version. This problem can be fixed if a part of the encoder implementation is frozen to keep rsyncable output stable across xz versions.

Embedded .xz decompressors

Embedded .xz decompressor implementations like XZ Embedded don't necessarily support files created with integrity check types other than none and crc32. Since the default is --check=crc64, you must use --check=none or --check=crc32 when creating files for embedded systems.

Outside embedded systems, all .xz format decompressors support all the check types, or at least are able to decompress the file without verifying the integrity check if the partic? ular check is not supported.

XZ Embedded supports BCJ filters, but only with the default start offset.

EXAMPLES

Basics

Compress the file foo into foo.xz using the default compression level (-6), and remove foo if compression is successful:

```
xz foo
```

Decompress bar.xz into bar and don't remove bar.xz even if decompression is successful:

```
xz -dk bar.xz
```

Create baz.tar.xz with the preset -4e (-4 --extreme), which is slower than the default -6, but needs less memory for compression and decompression (48 MiB and 5 MiB, respectively):

```
tar cf - baz | xz -4e > baz.tar.xz
```

A mix of compressed and uncompressed files can be decompressed to standard output with a single command:

```
xz -dcf a.txt b.txt.xz c.txt d.txt.lzma > abcd.txt
```

Parallel compression of many files

On GNU and *BSD, find(1) and xargs(1) can be used to parallelize compression of many files:

```
find . -type f \! -name '*.xz' -print0 \
    | xargs -0r -P4 -n16 xz -T1
```

The -P option to xargs(1) sets the number of parallel xz processes. The best value for the -n option depends on how many files there are to be compressed. If there are only a couple of files, the value should probably be 1; with tens of thousands of files, 100 or even more may be appropriate to reduce the number of xz processes that xargs(1) will eventually create.

The option -T1 for xz is there to force it to single-threaded mode, because xargs(1) is used to control the amount of parallelization.

Robot mode

Calculate how many bytes have been saved in total after compressing multiple files:

```
xz --robot --list *.xz | awk '/^totals/{print $5-$4}'
```

A script may want to know that it is using new enough xz. The following sh(1) script checks that the version number of the xz tool is at least 5.0.0. This method is compatible with old beta versions, which didn't support the --robot option:

```
if ! eval "$(xz --robot --version 2> /dev/null)" ||
        [ "$XZ_VERSION" -lt 50000002 ]; then
    echo "Your xz is too old."
fi
unset XZ_VERSION LIBLZMA_VERSION
```

Set a memory usage limit for decompression using XZ_OPT, but if a limit has already been set, don't increase it:

```
NEWLIM=$((123 << 20))  # 123 MiB
OLDLIM=$(xz --robot --info-memory | cut -f3)
if [ $OLDLIM -eq 0 -o $OLDLIM -gt $NEWLIM ]; then
    XZ_OPT="$XZ_OPT --memlimit-decompress=$NEWLIM"
    export XZ_OPT
fi
```

Custom compressor filter chains

The simplest use for custom filter chains is customizing a LZMA2 preset. This can be useful, because the presets cover only a subset of the potentially useful combinations of compression settings.

The CompCPU columns of the tables from the descriptions of the options -0 ... -9 and --ex?

treme are useful when customizing LZMA2 presets. Here are the relevant parts collected from those two tables:

Preset	CompCPU
-0	0
-1	1
-2	2
-3	3
-4	4
-5	5
-6	6
-5e	7
-6e	8

If you know that a file requires somewhat big dictionary (for example, 32 MiB) to compress well, but you want to compress it quicker than xz -8 would do, a preset with a low CompCPU value (for example, 1) can be modified to use a bigger dictionary:

```
xz --lzma2=preset=1,dict=32MiB foo.tar
```

With certain files, the above command may be faster than xz -6 while compressing signifi? cantly better. However, it must be emphasized that only some files benefit from a big dictionary while keeping the CompCPU value low. The most obvious situation, where a big dictionary can help a lot, is an archive containing very similar files of at least a few megabytes each. The dictionary size has to be significantly bigger than any individual file to allow LZMA2 to take full advantage of the similarities between consecutive files.

If very high compressor and decompressor memory usage is fine, and the file being com? pressed is at least several hundred megabytes, it may be useful to use an even bigger dic? tionary than the 64 MiB that xz -9 would use:

```
xz -vv --lzma2=dict=192MiB big_foo.tar
```

Using -vv (--verbose --verbose) like in the above example can be useful to see the memory requirements of the compressor and decompressor. Remember that using a dictionary bigger than the size of the uncompressed file is waste of memory, so the above command isn't use? ful for small files.

Sometimes the compression time doesn't matter, but the decompressor memory usage has to be kept low, for example, to make it possible to decompress the file on an embedded system. The following command uses -6e (-6 --extreme) as a base and sets the dictionary to only

64 KiB. The resulting file can be decompressed with XZ Embedded (that's why there is --check=crc32) using about 100 KiB of memory.

```
xz --check=crc32 --lzma2=preset=6e,dict=64KiB foo
```

If you want to squeeze out as many bytes as possible, adjusting the number of literal con? text bits (lc) and number of position bits (pb) can sometimes help. Adjusting the number of literal position bits (lp) might help too, but usually lc and pb are more important.

For example, a source code archive contains mostly US-ASCII text, so something like the following might give slightly (like 0.1 %) smaller file than xz -6e (try also without lc=4):

```
xz --lzma2=preset=6e,pb=0,lc=4 source_code.tar
```

Using another filter together with LZMA2 can improve compression with certain file types. For example, to compress a x86-32 or x86-64 shared library using the x86 BCJ filter:

```
xz --x86 --lzma2 libfoo.so
```

Note that the order of the filter options is significant. If --x86 is specified after --lzma2, xz will give an error, because there cannot be any filter after LZMA2, and also because the x86 BCJ filter cannot be used as the last filter in the chain.

The Delta filter together with LZMA2 can give good results with bitmap images. It should usually beat PNG, which has a few more advanced filters than simple delta but uses Deflate for the actual compression.

The image has to be saved in uncompressed format, for example, as uncompressed TIFF. The distance parameter of the Delta filter is set to match the number of bytes per pixel in the image. For example, 24-bit RGB bitmap needs dist=3, and it is also good to pass pb=0 to LZMA2 to accommodate the three-byte alignment:

```
xz --delta=dist=3 --lzma2=pb=0 foo.tiff
```

If multiple images have been put into a single archive (for example, .tar), the Delta fil? ter will work on that too as long as all images have the same number of bytes per pixel.

SEE ALSO

xzdec(1), xzdiff(1), xzgrep(1), xzless(1), xzmore(1), gzip(1), bzip2(1), 7z(1)

XZ Utils: <https://tukaani.org/xz/>

XZ Embedded: <https://tukaani.org/xz/embedded.html>

LZMA SDK: <http://7-zip.org/sdk.html>