# Documentation for LZEXE, EXE file packer.

The ecm fork of LZEXE v0.91 is based on the 2025 May free software release under the MIT license. Refer to the LICENSE file for full attribution and usage conditions.

Hardware: PC and compatibles, 80286 or 80386 microprocessor recommended for greater execution speed. Memory required: 128 KiB minimum to run LZEXE.

This document has been compiled on 2025-07-30.

# Contents

# Section 1: Introduction

This software compresses EXE files, that is, EXEcutable files from the 8086 DOS PC world. But you could tell me that a lot of software compresses EXE files better than this one, such as the excellent PKZIP or LHARC. But the advantage of this program is that your EXE files once compressed can be launched! And the depacking is so fast that for virtually all files, this depacking time is negligible! In addition, the depacker does not use any additional disk space or RAM on a virtual disk: it only uses the RAM normally reserved for the unpacked EXE file. In addition, I have greatly optimized my depack algorithm in speed but also in efficiency: EXE files are almost as small as the corresponding ZIP files and much more compact than the old ARC files.

# Section 2: Using LZEXE

## 2.1  Running LZEXE

It's very simple: just type in DOS:

```
LZEXE [switches] filename[.EXE]
```

where 'filename' is the name of the EXE file you want to compress. The .EXE extension is added by default. The compressed file is created in the default directory. The 'switches' part is optional and may hold any number of switches. Switches are indicated by a leading slash. Refer to section 2.2 for the meaning of the switches.

Warning! Some files are EXE only by their name: in fact, for DOS, it is not the .EXE extension that characterizes this type of file, it is the fact that at the beginning there are the letters 'MZ' or 'ZM' followed by a few bytes that indicate the length of the file, the memory size that it occupies, etc. So some people do not hesitate to rename COM files to EXE, and this explains why LZEXE refuses certain EXE files that are just renamed COM files.

But there is a method to make LZEXE accept COM files: just use the LZEXE companion tool COMTOEXE which converts COM files to EXE files.

For added security, LZEXE does not erase your old EXE file: it renames it to *.OLD. In addition, it creates the temporary file LZEXE.TMP which is only renamed to *.EXE after packing is complete.

## 2.2  Switches for LZEXE application

The following switches are supported:

`/1`

   Choose old v0.91 stub format

`/2`

   Choose new stub format (default)

`/I`

   Inline getbit code alike old stub (faster)

`/J`

   Do not inline getbit code (slower, default)

5

`/S`

> Stop optimisation, always use LZE3 equivalent

`/O`

> Optimise to drop relocs or segment change (default)

`/L`

> Allow output file that is not smaller than input

`/-`

> No more switches, filename follows

`//`

> Same as `/-`

`/#XX`

> Force LZ signature letters to XX (for debugging)

### 2.2.1  Switch `/1`

Chooses to pack with the old v0.91 stub format. This generates a file largely compatible with the output of LZEXE v0.91 albeit not identicalised to it. The length of the depacker stub and the placement of its variables do exactly match.

### 2.2.2  Switch `/2`

Chooses to pack with the new stub format. (This is the default.) The current format is known as LZX0, and comes in 8 variants. This format is not compatible with any older depackers expecting the LZEXE v0.90 or v0.91 formats.

### 2.2.3  Switch `/I`

Inline getbit code alike old stub. Inlining the getbit code makes depacking faster at the expense of some code size. In the old format, the code was inlined.

### 2.2.4  Switch `/J`

Do not inline getbit code. This is slower than inlining but saves some space. (This is the default.)

### 2.2.5  Switch `/S`

Stop optimisation, always use LZE3 equivalent. This disables the dropping of the relocation table if empty, and of the segment change code if the uncompressed image size is below 40 KiB.

### 2.2.6  Switch `/O`

Optimise to drop relocs or segment change. Different variants of the LZX0 depacker stub can be used to optimise stub size. This switch enables two different optimisations. The LZEXE application will detect if either or both of these optimisations can be used. (This is the default.)

### 2.2.7 Switch **/L**

Allow output file that is not smaller than input. Usually LZEXE will delete its temporary output file if it is at least as large as the input file. With this switch, it will only warn about the condition but still write to the destination file.

### 2.2.8 Switch **/-**

No more switches, filename follows. This switch indicates that the next input on the command line is a filename, even if it starts with a dash or slash. This avoids misdetecting such names as switches.

### 2.2.9 Switch **//**

Same as /-.

### 2.2.10 Switch **/#xx**

Force LZ signature letters to xx (for debugging). The xx can be replaced by any two printable ASCII bytes. The specified bytes are appended to the LZ signature. This overrides the LZ91 or LZX0 signatures usually chosen by the current LZEXE. This switch should not be used lightly and is intended only for debugging LZEXE. It can lead to data corruption if misused.

# Section 3: Tips for use

For some files, compression may not work for several reasons:

- The file you specified is not a real EXE. Solution: use COMTOEXE.EXE

- The relocation table is too large. To understand this message, it's necessary to understand the internal structure of an EXE file: such a file can span multiple segments, unlike COM files. This is why it needs a table of values that indicates in which segment branches or subroutine calls are made, for example. And if the file is very long, this table can be very bulky and prevent the packer from working. However, I have allowed 16,000 relocation addresses, which should be sufficient for all EXE files, even the largest.

- The file you specified has already been compressed with LZEXE.

  Note that another file packer exists: EXEPACK.EXE from Microsoft. But it is far less efficient than mine, and even if your EXE file is already packed with this program, LZEXE will still be able to pack a lot. But in this case, you'll be presented with a warning message, because there's another LZEXE companion tool: UPACKEXE, which allows you to unpack these files, and thus the gains are even greater.

- The packing was not efficient enough and wasted space on disk: Yes, it can happen, but generally with small EXE files (less than 2 KiB). Otherwise, you can almost always win a few bytes. If the output file is at least as large as the input file, LZEXE will abort the operation and delete the temporary output file, unless the `/L` switch was specified.

- The EXE file contains internal overlays: these are pieces of program files that are in the EXE file but are loaded only when the main program needs them. LZEXE cannot pack them because it would be necessary to modify the loading routines which are in the main program, and unfortunately these routines depend on the compiler and programmer. In version 0.91, LZEXE warns you about their presence. But in some cases, the difference between the length of the EXE file on disk and the length of the loaded code is minimal (less than 1024 bytes): in this case, you can still choose to pack because these may be "marks" left by some compilers.

- In the ecm fork of LZEXE, a bug in LZEXE v0.91's relocation table format is fixed. The format cannot cope with a relocation at the very beginning of the executable program image, or more than one relocation at the exact same address. If the `/1` switch is used, indicating to use LZ91 format, the new LZEXE will detect executables with these and reject them rather than producing a depacker that crashes or corrupts random memory. If the LZX0 format is used these unusual relocations are fully supported.

- Some invalid EXE files are rejected. This includes files with zero pages, zero or negative sized images (too large header size), or relocations addressing space beyond the end of the image.

More serious is that some compressed EXE files will "crash" the machine:

- If the EXE program performs a test on its size or integrity on disk (this is very rare).

- If it contains overlays, which must be loaded afterwards and therefore must occupy fixed positions in the file.

- For programs that run under Windows (from Microsoft): these are not real 8086 DOS EXEs, so they will refuse to work properly when packed.

(This list is not exhaustive.)

Less serious: Some programs have configuration options that modify the EXE file (Turbo Pascal for example). In this case, you must first configure the program, then compress it and keep an uncompressed version so you can edit it.

# Section 4: The unlzexe utility

The companion tool unlzexe can reverse the LZEXE compression and generate an unpacked executable. It can handle all file formats generated by this version of LZEXE, except if the header signature was changed using the /# switch (refer to section 2.2.10).

Note that the exact header size and the order and addressing in the relocation entries may differ as compared to the original file before LZEXE compression. Additional data such as overlays or lDOS iniload's entries cannot be restored either, as they are not retained by LZEXE. The memory allocation and image size should be reproduced exactly. The relocation entries should lead to the same result, but as mentioned may be re-ordered and use different segmented addresses.

Compressing the unlzexe output using LZEXE (same version with the same switches) should produce exactly the same compressed file as the unlzexe input.

## 4.1 unlzexe debugging flags

The unlzexe tool accepts flags from the DEBUG environment variable. (%DEBUG% on DOS, $DEBUG on Linux.) The following flags are accepted in a mask:

1

    Enable all debugging output.

2

    If alloc delta field is present (not in current LZX0 format) then display both the old-style (unlzexe v0.7) and new-style resulting minimum allocation. These should match.

4

    Enable listing all code snippet patterns info. This also allows to determine which optimisations of the LZX0 format stub were used by LZEXE.

8

    Enable dump of MZ EXE header of input and output. The display is in both hexadecimal and decimal.

16

    Enable listing of input and output file sizes. The display is in both hexadecimal and decimal.

# Section 5: COMTOEXE utility

This program converts a COM (or BIN) file into an EXE file. It is the ideal complement to LZEXE since, thanks to COMTOEXE, LZEXE can also compress COM files.

Syntax:

```
COMTOEXE [switches] filename[.COM] [filename2[.EXE]]
```

where 'filename' is the name of the COM file to be converted. The COM extension is automatically added. The COM file is not deleted for added security.

By specifying 'filename2', you can specify another name for the EXE file.

The 'switches' part is optional and may hold any number of switches. Switches are indicated by a leading slash. Refer to section 5.1 for the meaning of the switches.

Some additional notes:

- Some COM files may not work after conversion, because some programs need to know the exact structure of the file they support (such as COMMAND.COM). Converting COM to EXE changes this structure slightly.

- For the /0 switch's and /1 switch's stub the resulting EXE file needs at least 64 KiB to load. If there is a smaller UMB and LOADHIGH is used, the resulting file will not load into the smaller UMB even if it may be large enough to hold the executable image. The EXE file's minimum allocation is set so that at least 64 KiB must be allocated to the process memory block.

- Relatedly, for /0 and /1 stubs, the stack pointer is always equal to FFFEh when the control flow is transferred to the original code.

- In the ecm fork, COMTOEXE adds a stub to the resulting EXE code. (This is the /1 switch operation.) This stub writes a zero word on the stack, sets SI to 100h, and finally jumps to the original entrypoint at 100h. The word on the stack may be expected by COM files so that they can use RETN to terminate.

- There is a size limit to the files accepted by COMTOEXE.

## 5.1 Switches for COMTOEXE

The following switches are supported:

/0

> Choose old-style no stub operation

/1

Choose new-style stub pushing zero (default)

`/2`

Choose stub that expands SP dynamically

`/A=num`

For /2 stub: Allocate at least `num` bytes after image

`/P=num`

For /2 stub: Set minimum SP offset before stub runs

### 5.1.1 COMTOEXE switch `/0`

Choose old-style no stub operation. This is the closest to the reverse of exe2bin. It prepends the executable image with a 32-byte MZ EXE header, followed by the literal flat-format image. No stub is appended. The initial CS:IP is equal to PSP:100h.

The minimum allocation is set so that at least 64 KiB are allocated to the process memory block, and the initial SS:SP is equal to PSP:FFFEh. However, this stack element is not initialised to a zero word.

### 5.1.2 COMTOEXE switch `/1`

Choose new-style stub pushing zero (default). This option causes COMTOEXE to prepend the 32-byte MZ EXE header and append an 8-byte stub behind the flat-format image. The initial CS is equal to PSP while the initial IP points to the stub.

The minimum allocation is set so that at least 64 KiB are allocated to the process memory block, and the initial SS:SP upon running the original entrypoint is equal to PSP:FFFEh. This stack element is initialised to a zero word by the stub.

### 5.1.3 COMTOEXE switch `/2`

Choose stub that expands SP dynamically. A 32-byte stub is appended behind the image. This stub reads the actual size of the process memory block from the `word [PSP:2]` and sets up the SP so that SS:SP points either at PSP:FFFEh or at the last word of the process memory block, whichever is smaller. The stack element pointed to is also initialised to a zero word by this stub.

The minimum allocation as well as initial SP in the MZ EXE header can be set up to allow a process memory block smaller than 64 KiB. Due to the stub, the initial SS:SP when running the image at PSP:100h will be adjusted to be as large as PSP:FFFEh, without requiring the program always be allocated a full 64 KiB.

### 5.1.4 COMTOEXE switch `/A=num`

For /2 stub: Allocate at least `num` bytes after image. The `num` parameter can be a decimal number or a hexadecimal number. Hex numbers are indicated by leading '`0x`' or trailing '`h`'. The specified size indicates how much space is allocated to the process memory block at least, excluding the size of the PSP, the image, and a 128-byte reservation for the stack.

If the switch /2 stub is not in use, this switch does not take effect.

### 5.1.5  COMTOEXE switch `/P=num`

For /2 stub: Set minimum SP offset before stub runs. The `num` parameter can be a decimal number or a hexadecimal number. Hex numbers are indicated by leading '`0x`' or trailing '`h`'. This switch indicates the lowest value that the EXE header's SP should take. The header's minimum allocation is set so as to allocate the stack space.

If this switch is absent or set to 0, COMTOEXE will use a heuristic to scan up to 128 bytes of the executable image. The scan detects an initial `JMP NEAR` or `JMP SHORT` and will follow them, as long as they point to within the first 32 KiB of the input file. The scan searches for a `CMP SP,imm16` instruction followed by a conditional short jump (`JA`, `JB`, `JAE`, or `JBE`). If found, the comparison's immediate is used to determine the minimum SP.

If the calculated minimum SP is smaller than the size indicated by the `/A=` switch plus PSP size plus image size plus 128, the effective minimum SP is adjusted to be at least this large.

If the switch /2 stub is not in use, this switch does not take effect.

# Section 6: UPACKEXE utility

This program fills a major gap in the field of EXE file compressors/decompressors: It allows you to decompress EXE files compressed with Microsoft's EXEPACK.EXE and then recompress them with LZEXE, so that the gains are much greater. Programs reduced with EXEPACK are very common: practically all programs created with MSC (Microsoft C) are compacted with it, surely to hide its lack of optimisation! But the algorithm used, although very fast, cannot be compared with that of LZEXE, which performs much better.

Additionally, and this is why I created this unpacker, EXEPACK compacts the EXE file's relocation table and makes it inaccessible to LZEXE, which can no longer compress it to its full capacity. Thus, LZEXE's performance is slightly slower.

Syntax:

```
UPACKEXE filename[.EXE]
```

where 'filename' is the name of the file to unpack. It is renamed *.OLD. The new EXE file is created in the current directory under the name UPACKEXE.TMP and is renamed at the end.

# Section 7: LZEXEDAT utility

The LZEXEDAT utility compresses a data file into a raw compressed stream of the LZEXE format (refer to section 8.2). It does not parse an MZ header and the output file also doesn't have any headers at this time.

The utility is used in the following way:

```
lzexedat [switches] infilename outfilename
```

The following switches are supported:

`-b`

> Run a breakpoint immediately before calling LZCOMP.

`-v`

> Display verbose output instead of a single line.

`-4`

> Use the 4 KiB window variant encoding. This encoding is incompatible with the default (8 KiB window) encoding. The recipient must know which encoding was used.

`-l`

> Use the long literal variant encoding. This encoding is incompatible with the default encoding. The recipient must know which encoding was used.

`-c`

> Use the compatible encoding which emits 0000h as the combined word that escapes the end of stream or segment change commands. This makes the output compatible to the `-l` format even if `-l` is not specified.

`-d`

> Depack rather than pack. The `-l` and `-4` switches must match what was used to pack the file given as the input file to the depacker, except that `-l` `-d` to the depacker will work also if `-c` was specified to the packer without `-l`.

`-h`

> Operate on lDOS heatshrink format file. This is only supported when depacking with the `-d` switch. An lDOS heatshrink file starts with a byte giving the -w parameter and a second byte giving the -l parameter, followed by the compressed stream. The -w parameter must not be above 13 for the depacker in lzexedat; this is checked by the depacker.

15

`-e`

> Currently unused but accepted.

`--`

> Stop parsing of switches, all subsequent parameters are filenames.

Note that an empty input file is encoded as if it contained a single NUL byte.

## 7.1 lzexedat.sh

This bash script is provided to resolve leading dotdot in pathnames and call dosemu2 to run LZEXEDAT. The filenames should be short but especially not contain any blanks. Switches must be specified before the filenames.

# Section 8: From a technical point of view (for those in the know!)

The compression algorithm I made is based on the famous Lempel Ziv method using a "circular" buffer (ring buffer) and a method of finding repetitions of byte sequences by trees. The coding of the position and length of the strings that repeat is optimized by an additional algorithm inspired by the Huffman coding method. Unpacked literal bytes are sent as is in the file. An additional compression algorithm (like "Adaptive Huffman" (see LHARC) or with Shanon-Fano trees (see PKZIP)) would have required a longer depacking time and above all a more efficient depacker that is more complex and longer, which could actually make the compressed EXE file longer.

The depacker is located at the end of the EXE file and is 395 bytes in size long for version 0.90 and 330 bytes for version 0.91. In the ecm fork the depacker is between 208 and 305 bytes in size. The depacker must:

- Check the CRC to make sure no one has modified it (useful against viruses). If yes, display the message: 'CRC Error'. This option was removed in LZEXE v0.91 because it unnecessarily lengthens the file EXE and the depack time. Furthermore, the CRC check was only performed on the depacker.

- Move to the top of RAM, then move the compressed data to leave some space for the EXE image.

- Depack the code, checking that it is correct, and also adjust segments if you exceed 64 KiB (which caused me problems in terms of speed).

  In the LZX0 format of the ecm fork, the segment change handling can be omitted in case the original executable image was smaller than 40 KiB, saving some space for otherwise unused code in the depacker.

- Depack the relocation table, and update the relocation addresses in the EXE image. This is where LZEXE v0.91 was changed: the relocation table is much better compressed.

  In the LZX0 format of the ecm fork, the relocation table changed slightly. This supports a wider range of relocation entries.

  Further, LZX0 format can be used with the relocation table handling omitted in case the original EXE file had an empty relocation table.

- Before the subsequent transfer, set up AX (FCB drive validity) and DS, ES (both point to the PSP) the same way we received them from the DOS's program loader.

- Launch the program by updating CS,IP,SS,SP

That's all!!!

This depacker is a small masterpiece of 8086 assembler programming in itself: needless to say, it took quite a long time to develop. But the packer also posed quite a few problems for me, particularly when it came to updating all the pointers that the depacker uses later.

## 8.1 Optimisations in the ecm fork

I optimised the depacker some. In addition to the universal optimisations, LZX0 format can select one of eight variants of the depacker:

- With or without relocation table.

- With or without segment change handling.

- With getbit handling inlined or in a function.

The `/O` switch enables the optimisation of the relocation table and segment change code, whereas the `/S` switch stops these optimisations. The `/J` switch disables inlining getbit whereas the `/I` switch enables it.

## 8.2 Format of the LZEXE compressed data

The format of the LZEXE compressed data is a byte-based stream. The first word loaded from the stream is a tag word of bits. The tag bit counter is initialised to 16. A tag bit is consumed by shifting the tag word right and reading the shifted-out bit as the tag bit output. When the 16th bit is consumed, a new tag word of bits is loaded, and the tag bit counter is reset to 16.

The following bit sequences in the tags are known:

1

A literal byte command. The byte is read from the compressed stream.

0 0

A short (2 to 5 bytes) match command, with 8 bits of distance. Two more tag bits are read, the first being the high bit for the length and the second being the low bit. These two bits form a number in the range 0 to 3. After reading the two tag bits, 2 is added to the read value to retrieve a length in the range 2 to 5 bytes. Then one byte is read from the compressed stream. It is interpreted as a two's complement negative 8-bit number (0FFh means -1, down to 00h means -256) giving the distance.

0 1 length=001b to 111b

A displacement/length combined word is read. The low three bits of the high byte indicate the length value. Values 1 to 7 indicate a medium (3 to 9 bytes) match command, 2 is added to the read value. The low byte and the high five bits of the high byte indicate the two's complement negative 13 bits distance (0F8FFh means -1, 0000h means -8192).

0 1 length=000b byte=00h

The combined word's length is all-zeroes. A subsequent byte is read from the compressed stream. If it is zero, this is a marker for the end of the compressed stream. The distance is ignored.

0 1 length=000b byte=01h

> After the combined word's length is all-zeroes, the subsequent byte read is equal to one. This is a marker to normalise pointers. The distance is ignored. This marker should be placed at about every 40 KiB of uncompressed input data by the compressor. As 40 KiB must compress to 40 KiB / 8 bit/byte * 9 encoded bit/literal byte = 45 encoded KiB or less, both the source pointer and destination pointer needn't be normalised other than in response to this marker.

> The source pointer can be normalised in the naive way: Isolate the low 4 bits in si and shift right the high 12 bits from si by 4 bits, then add this value to ds.

> The destination pointer must be normalised differently. It must be handled so that di ends up in the range 8 KiB to 13 KiB. The original online depacker will isolate the low 4 bits of di and add 2000h to them (resulting in 8192 to 8207). The high 12 bits from di are shifted right by 4 bits, then 200h is subtracted from this value (which cannot underflow), then this value is added to es.

0 1 length=000b byte > 01h

> This is a long match command. The last byte read indicates a value in the range 2 to 255. One is added to this byte, leading to a length of 3 to 256. The distance of -1 to -8192 is used in the same way as for a medium match command.

After any command other than the end marker, the depacker loops back to the main depack loop which again reads the first tag bit to decode the next command.

The match commands may match with a negative match distance the absolute value of which is below the match length. In this case the depacker has to process the match command so that the initial prefix is replicated into the depacked output as often as needed to fulfil the command.

The very first command always has to be a literal command, so the very first bit (lowest bit of first byte) of the compressed stream is always a 1. Technically, the format can encode an empty output file in which case the first command is not a literal command. However, in practice an empty input file happens to be encoded into a literal command containing a NUL byte, followed by the end of stream command. (The first command may be a long literal command if the -l variant format is in use. In this case the compressed stream starts with a 0 bit.)

### 8.2.1  4 KiB variant format

The LZEXEDAT utility supports encoding using a variant on the window size. This is selected by passing the -4 switch to it. The window size variants are incompatible, and cannot be distinguished easily from inspecting the compressed stream.

In this variant, the match window that can be addressed by medium and long match commands is only 4 KiB rather than 8 KiB. The displacement/length combined word only uses the low byte and the high four bits (rather than five) of the high byte to encode the 12-bit distance. Therefore, the length is encoded in four (rather than three) bits.

Similarly to the original variant, a length of zero (0000b) indicates an escape for long match, segment change, or end of stream commands. The length for medium match commands is encoded as 0001b to 1111b (1 to 15), so a medium match command can have a length of 3 to 17 bytes.

### 8.2.2  Long literal variant format

The LZEXEDAT utility supports encoding in yet a different variant. This is selected by passing the -l switch to it. Again, the long literal variant is incompatible with the short literal one, and they cannot be distinguished easily from inspecting the compressed stream.

In this variant, the encoding for a segment change always encodes a 13-bit or 12-bit (depending on -4) all-zeroes displacement. Nonzero displacements with the combined word's length as zero and the escaped byte equal to 1 instead do not encode a segment change. Rather, the displacement is taken to mean a counter of literal bytes that follow in the compressed stream. It must be at least 26, because 25 or fewer literals can be encoded more efficiently by repeatedly using the single-literal command.

Currently the packer will use at most 512 literals in one long literal command. It is assumed that a segment change will be inserted appropriately, but other than that the packer is allowed to encode up to 8191 or 4095 literal bytes in one long literal command. The limit is due to the displacement encoding scheme.

The long literal format requires a small alteration in the depacker. On large files ($> 100$ kB) usually there will be some savings for the size of the compressed stream. On small files ($< 4$ KiB) the savings may be negligible so it is usually better to not choose to enable long literals then. The file source.txt of lDebug has been observed to compress to 250 bytes with lzexedat -4 but 251 bytes with lzexedat -4 -l. This appears to be due to an unfortunate allocation of tag bits leading to encoding one more tag word, almost empty, at the end. This is an outlier and the effect is likely limited to no more than 2 bytes of added file size.

### 8.2.3  Compatible format

The -c switch to LZEXEDAT indicates use of the compatible format. A file packed using -c is compatible to depackers assuming the long literal format, regardless of whether -l was specified.

The change due to -c is that both the End Of Stream (EOS) command and the segment change command are encoded using 0000h as the combined word. Absent the -c switch, EOS commands are encoded with a combined word of 0F000h. Absent both the -c and -l switches, segment change commands are also encoded with a combined word of 0F000h.

# Section 9: LZEXE version 0.91 and other compressors

PKARC (latest version): LZEXE performs much better, as "crunching" (aka shrinking for PKZIP) is an outdated algorithm...

PKZIP v0.92: LZEXE does better in almost all cases.

PKZIP v1.02: On large files, LZEXE does better. otherwise, the difference is quite small.

LHARC v1.01: It does better than LZEXE with "freezing" on small files.

LARC: LZEXE does better.

Important Notes:

- You can't really compare what LZEXE does with other packers since in the EXE files compressed by my software there is also a depacker that launches the EXE by itself. The other compressors can make "self-extracting" files, but they depack to disk, are slow and add several tens of KiB to the compressed files (except for LARC and LHARC which only add 1 or 2 KiB, but which only depack on disk unfortunately).

- In almost all cases, the compressors I mentioned will not be able to further pack a file already reduced with LZEXE, which shows its efficiency. Only LHARC manages to save a few bytes.

UPX: `upx-ucl --8086 --lzma` does about 6% better than LZEXE. Like LZEXE it produces files with an online depacker stub in the compressed executable file.

# Section 10: The future...

- I'm also thinking about an automatic documentation depacker like README.COM or LIST.COM, which would be very convenient, and maybe I'll make one.

- Finally, I hope to make a "universal" packer like PKZIP or LHARC, which is slower than LZEXE when it comes to depacking, but performs much better than them.

# Section 11: Warnings and wishes...

I hope that LZEXE and the EXE files compressed by it will be widely broadcast which will encourage me to make other versions more quickly...

I decline all responsibility in the event of loss of information caused by LZEXE. But rest assured, the algorithms are reliable and I don't think there are many bugs.

Warning! I do not recommend compressing and distributing commercial software protected by copyright: the authors may be displeased...

But if you're making a FREEWARE, SHAREWARE, or even a commercial program, nothing's stopping you from compressing it with LZEXE, and I even recommend it:

- Your EXE files will be smaller, and your compiled programs will look like they're made in assembler. What will your competitors say when they see programs that do the same thing as theirs but are 30% smaller? Plus, you'll be able to fit more programs on floppy disk (and hard disk), because we always need more mass storage...

- Compression is an excellent form of coding that can prevent unsavory people from modifying messages or seeing your secret algorithms unless they disassemble the depacker, which might not be very easy, I'm telling you!

There you go, hoping that this software will be useful to you and that it does not have too many bugs!

# Section 12: Evolution of versions

## 12.1  LZEXE ecm release 4 (2025-07-27)

- lzexedat allows to depack (-d) lzexedat- or heatshrink-compressed (-h) files. Re-uses the circular buffer depacker developed for the compressed Extensions for lDebug library, allowing to operate on file sizes > 64 KiB.

- Default to building the FreePascal builds rather than TurboPascal.

- lzexedat switches -4 and -l and -c for variant formats.

- Add lzexedat.sh script to run lzexedat.exe in dosemu2.

- Implement second name parameter to lzexe which gives the directory to which to write the temporary and final output files.

- Optimise depacker stub some more.

- Adjust minimum allocation in upackexe.

- In upackexe do not hardcode 512 bytes per header.

- Add upackexe debugging output by setting the %DEBUG% variable.

- Fix some calculations and displayed results in infoexe to work with amount of bytes exceeding 16 bits.

## 12.2  LZEXE ecm release 3 (2025-07-01)

- Add comtoexe stub 2 (/2 switch) and /A= and /P= switches to create a smart .EXE file which can run with below 64 KiB of memory but will dynamically extend SP to the actual size.

- Add comtoexe switches /0 and /1 for selecting no stub or short stub.

- Fix UPACKEXE if a relocation with offset 0FFFFh appears in the exepacked relocation table.

- Translate and update this manual.

- unlzexe dot mode: If second parameter is a lone dot, only calculate everything without writing to any file.

## 12.3  LZEXE ecm release 2 (2025-06-27)

- Add formats without inlined getbit function. This is purely a speed vs size tradeoff.

- Add debugging outputs to unlzexe, selected using the DEBUG environment variable.

- Make unlzexe able to restore the minimum allocation size of the original unpacked executable file, even if alloc delta field absent.

- Add formats without segment change handling.

- Add format without relocation table.

- Flush the pipeline for depacker's Self Modifying Code.

## 12.4 LZEXE ecm release 1 (2025-06-22)

- Do not calculate percentages with the 8087 FPU.

- Port all Pascal source texts to allow building with FreePascal, i8086 DOS target.

- Add alloc delta field to depacker stub so unlzexe works better.

- Detect invalid relocations (past image).

- Add new relocation table format which allows zero-difference relocations.

- Add new depacker, optimise it, and pass along the value in AX from the DOS program loader to the depacked program.

- Check for nonzero positive image size.

- If unknown LZ signature is found, display it.

- Make unlzexe compile for either 8086 DOS (gcc ia16) or amd64 Linux (gcc).

- Add unlzexe from LZEXE v0.91e, the former is Public Domain.

- Port packer to NASM.

- In comtoexe add a stub to set up a zero word on top of the stack so that programs depending on it won't break.

- Detect and warn about invalid amount last page bytes field (> 512) and display a hint on lDOS iniload if the low byte of this field equals 0EBh.

- Detect and reject zero in EXE header amount pages field.

- Translate most of the main LZEXE tool to the english language.

## 12.5 LZEXE ecm release 0 (2025-06-17)

- Reject zero-difference relocations when packing a file with the LZ91 format.

- Display remaining file size and percentage.

- Port depacker to NASM.

## 12.6 LZEXE v0.91 (1990-04-11)

- The hyphen '-' was not accepted in file names, this has been corrected.

- LZEXE signals the presence of internal overlays.

- LZEXE indicates whether the file has already been packed with EXEPACK by Microsoft.

- A bug in version 0.90 caused EXE files to take up too much memory: this has been fixed.

- Relocation table compression has been improved.

- CRC check has been removed.

- Decompressor size reduced from 395 to 330 bytes.

# Source Control Revision ID

hg 7a0bc7e19cbb, from commit on at 2025-07-30 19:46:52 +0200

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/lzexe/rev/7a0bc7e19cbb