lDOS boot documentation

2020--2025 by E. C. Masloch. Usage of the works is permitted provided that this instrument is retained with the works, so that any entity that uses the works is notified of this instrument. DISCLAIMER: THE WORKS ARE WITHOUT WARRANTY.

This document has been compiled on 2026-11-14.

Contents

Section 1: lDOS boot protocols 4
1.1 Sector to iniload protocol 4
1.1.1 File properties 4
1.1.2 Signatures 4
1.1.3 Load Stack Variables (LSV) 5
1.1.4 LSV extension - Command line 6
1.1.5 LSV extension - Extra 7
1.1.6 Pre-boot loader support 8
1.1.7 Memory map 9
1.1.8 Load filename in the boot sector 9
1.1.9 Query patch support 9
1.1.10 lDOS boot error condition letters 10
1.2 Iniload to payload protocol 11
1.2.1 Stack 11
1.2.2 Extended BIO Parameter Block (EBPB) 12
1.2.3 Load Stack Variables (LSV) 12
1.2.4 Load Data 1 (LD) 12
1.2.5 Load Command Line (LCL) 13
1.3 Application mode protocol 14
1.4 Device mode protocol 15
1.5 Second stage FAT32 protocol, FSIBOOT 15
Section 2: DRLoad boot protocols 17
2.1 Sector to drload protocol 17
2.2 DRLoad to payload protocol 17
Section 3: Other documented boot load protocols 18

3.1 ROM-BIOS to sector load protocol 18
3.1.1 ROM-BIOS load - Assumptions 18
3.2 MBR loader to sector load protocol 18
3.2.1 lDOS partition info detection 18
3.2.2 Microsoft partition type detection 19
3.3 Sector to FreeDOS load protocol 19
3.3.1 FreeDOS load - Assumptions 19
3.3.2 FreeDOS load extension - Command line 19
3.4 Sector to Enhanced DR-DOS load protocol 20
3.4.1 Enhanced DR-DOS second file 20
3.5 Sector to MS-DOS v6 load protocol 21
3.5.1 MS-DOS v6 load - Assumptions 21
3.6 Sector to IBM-DOS load protocol 22
3.7 Sector to MS-DOS v7 load protocol 22
3.7.1 MS-DOS v7 load - Assumptions 23
3.7.2 Multiboot v1 and Multiboot v2 specification 23
Section 4: lDOS iniload modes 24
Source Control Revision ID 25

Section 1: lDOS boot protocols

1.1 Sector to iniload protocol

The iniload kernel is loaded to an arbitrary segment. The segment must be at least 60h. Common choices are 60h, 70h, and 200h. At least 1536 bytes of the file must be loaded. Current loaders will load at least 8192 bytes if the file is as large or larger than that. (If the instsect option `/4 PREFIX` is used, only up to 4 KiB may be loaded.) The entrypoint is found by applying no segment adjustment (0) and choosing the offset 400h (1024).

The segment choice of 200h (8 KiB) avoids any possible sector reads across a 64 KiB ISA DMA boundary, even with the largest supported sector size (also 8 KiB).

1.1.1 File properties

The file must be at least 4096 bytes long. This is now required, beyond the former lower bound of 1536 bytes, to support an optimisation of the FAT12 and FAT16 boot sector loaders. The lDebug loader and the FAT32+FSIBOOT loader currently retain the 1536 bytes limit.

The file may allow multi-use as a flat .COM format executable, flat .SYS format device driver, or MZ .EXE format executable and/or device driver. It is also valid to append arbitrary sized data, such as a .ZIP or .7Z archive, or an Extension for lDebug using the eldapend tool. However, to support entry as a FreeDOS or EDR-DOS kernel file using the original FreeDOS loaders, the entire file must be no larger than 128 KiB. To detect EDR-DOS entry the file must be at least 32 KiB in size. Loading as a device driver from config.sys may also check the entire file size on some DOS versions.

The file needs to be placed in the root directory for the boot sector loaders. The lDebug loader allows to load a file from any subdirectory and this is also allowed. (This may interfere with attempts to find the load file though.) The file may be fragmented in any part. The file data may be located anywhere in the file system. The supported cluster sizes should be between 32 Bytes and 2 MiB, inclusive. The sector size should be between 32 Bytes and 8 KiB, inclusive.

1.1.2 Signatures

At offset 1020 (3FCh) there is the signature '`lD`'. Behind that there are two bytes with printable non-blank ASCII codepoints. Currently the following signatures are defined:

'`lDOS`'

: lDOS kernel (used for lMS-DOS based kernel since 2025 February)

'`lDRx`'

: RxDOS kernel

‘lDFD’

FreeDOS kernel in fdkernpl stage

‘lDDR’

Enhanced DR-DOS single-file kernel in drkernpl stage

‘lDMS’

lMS-DOS single-file kernel in drkernpl stage (until 2025 February)

‘lDeb’

lDebug

‘lDDb’

lDDebug (debuggable lDebug)

‘lDbC’

lCDebug (conditionally debuggable lDebug)

‘lDTP’

lDOS test payload kernel (testpl.asm)

‘lDTW’

lDOS test result writer kernel (testwrit.asm)

‘lDLd’

lDOS pre-boot loader

‘lDMC’

lDOS Master Control Program (contains lDOS kernel, lCDebugX, and loader)

‘lDXX’

Signature not set (shouldn't be used)

1.1.3 Load Stack Variables (LSV)

Under this protocol, the pointer ‘ss:bp’ is passed. It points to a boot sector with (E)BPB. ‘bp’ must be even for compatibility with older iniload (before 2023-March). The stack pointer must be at most ‘bp - 10h’. Below the pointed to location there live the Load Stack Variables. These follow this structure:

```
        struc LOADSTACKVARS, -10h
lsvFirstCluster:        resd 1
lsvFATSector:           resd 1
lsvFATSeg:              resw 1
lsvLoadSeg:             resw 1
lsvDataStart:           resd 1
        endstruc
```

lsvFirstCluster

(FAT12, FAT16) Low word gives starting cluster of file. High word uninitialised.

(FAT32) Dword gives starting cluster of file.

(else) Should be zero.

lsvFATSector

(FAT16) Low word gives loaded sector-in-FAT. -1 if none loaded yet. High word uninitialised.

(FAT32) Dword gives loaded sector-in-FAT. -1 if none loaded yet.

(FAT12, else) Unused.

lsvFATSeg

(FAT16, FAT32) Word gives segment of FAT buffer if word/dword [lsvFATSector] != -1.

(FAT12) Word gives segment of FAT buffer. Zero if none. Otherwise, buffer holds entire FAT data, up to 6 KiB.

lsvLoadSeg

Word points to segment beyond last loaded paragraph. Allows iniload to determine how much of it is already loaded.

lsvDataStart

Dword gives sector-in-partition of first cluster's data.

1.1.4 LSV extension - Command line

An LSV extension allows to pass a command line to the kernel. The base pointer must be at least '`114h`' then. The stack pointer must be at most '`bp - 114h`' then. This follows the structure like this:

```
lsvclSignature          equ "CL"
lsvclBufferLength       equ 256

        struc LSVCMDLINE, LOADSTACKVARS - lsvclBufferLength - 4
lsvCommandLine:
.start:         resb lsvclBufferLength
.signature:     resw 1
lsvExtra:               ; word
.partition:     resb 1  ; byte
.flags:         resb 1  ; byte
        endstruc
```

lsvCommandLine.start

Command line buffer. Contains zero-terminated command line string.

lsvCommandLine.signature

Contains the signature value ‘CL’ if command line is given.

lsvExtra

Used internally by iniload. Space for this must be reserved when passing a command line.

If no command line is passed then either the stack pointer must be ‘bp - 10h’, or ‘bp - 12h’, or the word in the lsvCommandLine.signature variable (word [ss:bp - 14h]) must not equal the string ‘CL’.

- dosemu2's RxDOS.3 support sets ‘sp = bp - 10h’
- ldosboot boot.asm (FAT12/FAT16) loader makes sure not to pass the variable with the content "CL". Refer to placeholder and DIRSEARCHSTACK_CL_FIRST uses in the source.
- ldosboot boot32.asm (FAT32) loader uses the variable for an ‘entries per sector’ value which is always a power of two and always below-or-equal 100h.
- lDebug with protocol options cmdline=0 push_dpt=0 sets ‘sp = bp - 10h’

1.1.5 LSV extension - Extra

The lDOS iniload entry at 400h in the file contains the following 9 bytes of code:

```
ldos_entry:
        cli
        cld
        xor ax, ax
        push ax         ; push into lsvExtra if sp -> LSV
        mov word [bp + lsvExtra], ax
        push ax         ; push into lsvCommandLine if sp -> LSV
```

This assembles to:

```
400 FA
401 FC
402 31C0
404 50
405 8946EE
408 50
```

The same structure as shown in section 1.1.4 is expected by this code:

```
lsvclSignature          equ "CL"
lsvclBufferLength       equ 256

        struc LSVCMDLINE, LOADSTACKVARS - lsvclBufferLength - 4
lsvCommandLine:
.start:         resb lsvclBufferLength
.signature:     resw 1
lsvExtra:               ; word
.partition:     resb 1  ; byte
```

```
.flags:         resb 1  ; byte
        endstruc
```

The currently known flags for the lsvExtra.flags byte:

```
lsvefNoDataStart        equ 1
lsvefPartitionNumber    equ 2
lsvefPreserveLoader     equ 4
```

By detecting the 9-byte entry code snippet, one is able to avoid the zeroing of lsvExtra. To do this, ss:sp should be made to point at lsvCommandLine.signature or lower. The word lsvCommandLine.signature should be initialised appropriately. The ip should be set to 409h, ax to zero, and the Interrupt Flag and Direction Flag should be cleared. Entering the initial loader then will preserve lsvExtra.

1.1.6 Pre-boot loader support

There may be a signature, aligned on a 16-byte boundary, that reads '`LOADERSUPPORT00`'. After the 15 text bytes a flag byte follows, which currently must be all-zeroes. If this signature is present within the first 4096 bytes of the file, it indicates that:

- This initial loader loads an lDebug debugger.
- The initial loader LSV extra extension is supported.
- The debugger kernel mode entry supports the lsvefPreserveLoader flag.
- If the flag is set, the debugger will detect a hidden NLDR and call into it to preserve it and relocate it to the low position during the debugger's installation.

Hidden means that the NLDR is installed atop the Low Memory Area, either at the position indicated by the int 12h memory size or behind an EBDA. The NLDR header is of the following format:

```
        struc LOADERIMAGEIDENT
liiSignature:           resb 4  ; "NLDR"
liiVersion:             resb 2  ; "00"
liiChecksum:            resw 1  ; word sum of first 8 words = 0
liiAmountParagraphs:    resw 1
liiReserved:            resw 3
liiEntryQuerySize:      resw 1  ; 0 if not supported
        ; INP:  cs => NLDR image ident
        ;       ip = this word field content
        ;       ax = 60h
        ; OUT:  ax => behind memory requested (should be >= 60h)
        ; CHG:  es, ds, di, si, bx, cx, dx
        ; STT:  far called, UP, EI
liiEntryRelocate:       resw 1          ; 0 if not supported
        ; INP:  cs => NLDR image ident
        ;       ip = this word field content
        ;       ax = 60h
        ; OUT:  ax => relocated NLDR segment (usually 60h)
        ; CHG:  es, ds, di, si, bx, cx, dx
```

```
        ; STT:  far called, UP, EI
liiEntryDebug:          resw 1          ; 0 if not supported
        ; INP:  es = ds = cs = ss => relocated NLDR
        ;       ip = this word field content
        ;       sp = next word field content
        ; STT:  running as debuggee, UP, EI
liiStackPointer:        resw 1
        endstruc
```

The pointers in this structure are used by the debugger during its initialisation if the lsvefPreserveLoader flag was set.

1.1.7 Memory map

The initial loader part that is loaded must be loaded at above or equal to linear 00600h. The FAT buffer segment (if used) must also be stored at above or equal to linear 00600h. The stack (which should extend at least 512 bytes below '`ss:bp`') and boot sector (pointed to by '`ss:bp`', at least 512 bytes length) should also be stored at above or equal to linear 00600h.

There is an additional memory area, the Low Memory Area top reservation, which should be unused by the load protocol at handoff time but be at least 20 KiB in size. It is located below the usable Low Memory Area top. That is, directly below the EBDA, RPL-reserved memory, video memory, or otherwise UMA. This area is reserved in order to facilitate initial loader operation.

None of the memory areas may overlap. This does not include the FAT buffer in case it is uninitialised.

1.1.8 Load filename in the boot sector

The boot sector may be expected to contain a valid 8.3 format (blank-padded FCB) filename in the area of the boot sector starting behind the (E)BPB, extending up to below the boot sector signature word with value AA55h (at offset 510 in the boot sector). This name should not contain blanks other than trailing in the file name portion or trailing in the file extension portion. It should consist of printable ASCII codepoints. That is, byte values between 20h and 7Eh inclusive. It should not consist of eleven times the same byte value. Additional FAT Short File Name restrictions may be assumed.

Although a loader should not depend on this for crucial operation, it may want to detect the kernel name it was presumably loaded from for informational or optional purposes. The canonical implementation of this is currently the function '`findname`' in the `testpl.asm` test payload kernel. It is found within the ldosboot repo. This handling is based on the function of the same name in the instsect application. (The lDOS MCP debugger wants to know the filename it was loaded from, in order to access its appended ELD library.)

1.1.9 Query patch support

The ldosboot repo includes a patch Script for lDebug (.sld) file which allows to patch the initial loader stage. The patches concern handling of the CHS geometry detection, and whether LBA or CHS access is used. There are several legacy patch sites in which patch.sld can directly patch the initial loader's code.

However, the preferred way is to find the query patch sequence. It should appear within the first 1536 bytes, that is within the part of the initial loader that must be loaded. This is the sequence:

```
8A5640  mov dl, byte [bp + 40h]
B8xxyy  mov ax, yyxxh
84D2    test dl, dl
7902    jns @F
86C4    xchg al, ah
@@:
```

The immediate word of the `mov ax` instruction is to be patched. The sequence should be scanned for without regard as to what the current contents of this word are.

The following flag values are used:

- 01h Force CHS access, do not detect LBA support with 13.41
- 02h Force LBA access, do not detect LBA support with 13.41
- 04h Force use of BPB's CHS geometry, do not detect with 13.08
- 08h Force use of single-sector accesses to load kernel
- 40h Used by test result writer (testwrit). If this value is set for the load unit, then the test application will skip all query geometry and LBA detection. Instead of running the detection, the information as passed by iniload is used verbatim. This differs from either flag 1 or flag 2 being set in that LBA support is carried over from iniload, and is not forced off or on. As for the CHS geometry, this is equivalent to setting the flags 84h (force use of passed geometry). The flag 40h takes effect even if flag 80h is not set.
- 80h Used by lDebug and test result writer (testwrit). If this value is set for the load unit, then lDebug or testwrit will make use of the other flags set up for that unit.

  For lDebug, the corresponding flags will be saved in lDebug's load_unit_flags. This affects only the load unit (LD in lDebug terminology), which suffices to pass commands in the startup Script for lDebug.

  For testwrit, the geometry query and LBA detection will honour the flags 01h, 02h, and 04h just like iniload interprets them. These only take effect if flag 40h is clear.

The flag 01h takes precedence over 02h if both are set.

The low byte (xxh) is used in case the loader loads from a diskette unit, that is a unit number below 80h. The high byte (yyh) is used otherwise, in case the loader loads from a hard disk unit, that is a unit number above-or-equal 80h.

The patchini repo contains a C language tool called patchqry which allows to read and write the query patch site. Unlike the Script for lDebug, patchqry can operate on arbitrarily large files.

1.1.10 lDOS boot error condition letters

The native original lDOS boot sector loaders are designed to emit a single error condition letter and then a beep code upon detectable errors. After that they wait for a keypress using int 16h function 00h then run an int 19h call. The terse display comes down to saving code space in the loaders.

The following letters are defined:

V

Check value mismatch

F

File not found

E

Not enough file data

R

Disk read error

B

Bad chain / bad FS

M

Out of memory

I

FSIBOOT error

S

Fix sector size mismatch

C

Fix cluster size mismatch

N

Non-FAT load needs larger cluster size

1.2 Iniload to payload protocol

The payload is loaded to an arbitrary segment. The segment must be at least 60h. The entire payload must be loaded. The size of the payload is determined at iniload build time. The entrypoint is found by applying a segment adjustment and choosing an offset. The segment adjustment is specified at iniload build time by the numeric define `_EXEC_SEGMENT` (default 0), and the offset by the define `_EXEC_OFFSET` (default 0).

1.2.1 Stack

The stack must allocate at least 256 bytes of space below `ss:sp`. This space must not underflow, ie `sp` is at least 256 (100h). `sp` should not be higher than 8192 (2000h). `bp` is typically equal to `sp + 120h`. `bp` must be below or equal 7000h. The stack starting at `ss:0` up to below `ss:bp + 21Ch` should be in the area between the paragraphs indicated by the load top lower limit and memory top upper limit. (Refer to LOADDATA fields ldLoadTop and ldMemoryTop.) Subsequent stages like inicomp, fdkernpl, drkernpl, checkpl, nullpl, and kernels like lDebug and testpl may depend on these characteristics.

1.2.2 Extended BIO Parameter Block (EBPB)

Above the LSV, `ss:bp` points to an EBPB and surrrounding boot sector. Note that this is always a FAT32-style EBPB. If the filesystem that is loaded from is not FAT32, and is therefore FAT16 or FAT12, then the FAT16/FAT12 BPBN structure is moved up. It is placed where the FAT32 BPBN is usually expected. In this case, the entire boot sector contents behind the BPBN are also moved up by the size of the FAT32-specific fields. The FAT32-specific fields are filled with zeros, except for the FAT32 'sectors per FAT' field. It is filled with the contents of the FAT16/FAT12 'sectors per FAT' field.

Note that a FAT32 file system is determined by whether the FAT12/FAT16 'sectors per FAT' BPB field is zero. If it is zero, then the EBPB is used unchanged from the boot sector. If it is nonzero, the FAT12/FAT16 BPB is expanded into an EBPB as described here. To do the expansion, typically the BPB New fields of the FAT12/FAT16 BPB as well as the trailing boot sector contents (up to below offset 512) are moved up the 1Ch bytes to open the gap for use by the FAT32-specific EBPB fields. That means the boot sector trailer reaches up to below offset 21Ch (540).

1.2.3 Load Stack Variables (LSV)

Refer to section 1.1.3.

1.2.4 Load Data 1 (LD)

Below the LSV, iniload passes the LOADDATA (1) structure.

```
        struc LOADDATA, LOADSTACKVARS - 10h
ldMemoryTop:    resw 1
ldLoadTop:      resw 1
ldSectorSeg:    resw 1
ldFATType:      resb 1
ldHasLBA:       resb 1
ldClusterSize:  resw 1
ldParaPerSector:resw 1
ldLoadingSeg:
ldQueryPatchValue:
                resw 1
ldLoadUntilSeg: resw 1
        endstruc
```

ldMemoryTop

> Word. Segment pointer to behind usable memory. Points at the first of the EBDA, RPL-reserved memory, or video memory or otherwise UMA. Indicates how much memory may be used by a typical kernel. (lDebug detects the EBDA to move that below where it installs.)

ldLoadTop

> Word. Segment pointer to lowest lDOS boot memory in use. All memory between linear 600h and the segment indicated here is usable by the payload. Only the payload itself is stored in this area. The other buffers, stack, and structures passed by iniload must live above this segment.

ldSectorSeg

Word. Segment pointer to an 8 KiB transfer buffer. It is insured that this buffer does not cross a 64 KiB boundary. This may be needed by some disk units. The buffer is not initialised to anything generally.

ldFATType

Byte. Indicates length of FAT entry in bits. 12 indicates FAT12, 16 FAT16, 32 FAT32. It is planned to allow zero for non-FAT filesystems.

If this is 12 and lsvFATSeg is nonzero, that variable points to the buffered FAT, either as many sectors as fill 6 KiB or the amount sectors per FAT from the BPB, whichever is less.

If this is 16 and the low word of lsvFATSector is not FFFFh, then lsvFATSeg points to a single-sector buffer with the specified FAT sector (0 = first FAT sector) buffered.

If this is 32 then it is treated the same as 16 except the entire dword of lsvFATSector counts. In this case lsvFATSector equal to FFFF_FFFFh means nothing is buffered.

ldHasLBA

Byte. Only least significant bit used. Bit on indicates LBA extensions available for the load disk unit. Bit off indicates LBA extensions not available.

ldClusterSize

Word. Contains amount of sectors per cluster. Unlike the byte field for the same purpose in the BPB, this field can encode 256 (EDR-DOS compatible) without any masking. May be given as zero for non-FAT filesystems.

ldParaPerSector

Word. Contains amount of paragraphs per sector. Must be a power of two between 2 (32 B/s) and 200h (8192 B/s). May be given as zero for non-FAT filesystems.

ldLoadingSeg

Word. Internally used by iniload. Available for re-use by payload. However, ldQueryPatchValue re-uses the same field.

ldQueryPatchValue

Word. Passes the query patch value from the initial loader. This provides an opportunity to patch a well-known site in the initial loader to change its behaviour in some ways. Near the end of its operation, the initial loader passes along this value in this variable for the next stage to use.

ldLoadUntilSeg

Word. Internally used by iniload. Available for re-use by payload.

1.2.5 Load Command Line (LCL)

Below the LOADDATA structure, iniload passes the LOADCMDLINE structure.

```
lsvclBufferLength       equ 256

        struc LOADCMDLINE, LOADDATA - lsvclBufferLength
ldCommandLine:
.start:         resb lsvclBufferLength
        endstruc
```

This buffer is always initialised to an ASCIZ string. At most 255 bytes may be initialised to string data. At most the 256th byte is a zero.

If the first word of the buffer is equal to 0FF00h, that is there is an empty command line the terminator of which is followed by a byte with the value 0FFh, then no command line was passed to iniload. Currently lDebug can pass a command line to iniload when loading with its lDOS, RxDOS.2, RxDOS.3, FreeDOS, or EDR-DOS protocols. When iniload is loaded as a Multiboot1 or Multiboot2 specification kernel, it is also assumed that a command line can be passed.

1.3 Application mode protocol

An iniload payload can be loadable as a DOS application. In this case the iniload MZ executable header will point to the payload. The _IMAGE_EXE define to iniload will indicate support for this if it is enabled.

The entire payload will be loaded as a program image just behind the PSP. The entrypoint is determined by the _IMAGE_EXE_CS and _IMAGE_EXE_IP defines. These default to a -16:256 + 64 entrypoint. Minus 16 means the code segment will be equal to the PSP, just like for flat .COM style executables. The 256 in the instruction pointer calculation skips past the PSP. Thus, the default _IMAGE_EXE_IP will enter the payload at offset 64 counting within the payload.

Likewise _IMAGE_EXE_SS and _IMAGE_EXE_SP specify the initial stack. They default to -16:0FFFEh, mimicking flat .COM style load somewhat. (However, iniload doesn't get a chance to push a zero word onto this stack before the first payload is run. Adding a nullpl, checkpl, or inicomp stage can be used to provide this.)

_IMAGE_EXE_AUTO_STACK defaults to zero. If it is nonzero it indicates to use the automatic stack placement. Its value, which defaults to 2048 if defined as empty, determines how large the stack should be (at least). The explicit _IMAGE_EXE_SS and _IMAGE_EXE_SP are ignored if the automatic stack placement is enabled. The automatic stack is placed into its own segment, stretching from ss:0 to ss:sp. The stack pointer will match the value of _IMAGE_EXE_AUTO_STACK.

_IMAGE_EXE_MIN indicates how much memory to allocate to the process at least. It defaults to 65536 (64 KiB). It is actually used in the _IMAGE_EXE_MIN_CALC define to calculate the minimum allocation value. _IMAGE_EXE_MIN_CALC is not a numdef, but can be overridden from the assembler command line. This is the default content of this define:

```
%define _IMAGE_EXE_MIN_CALC \
(((_IMAGE_EXE_MIN \
- (payload.actual_end - payload) \
- 256 \
+ _IMAGE_EXE_AUTO_STACK) + 15) & ~15)
```

_IMAGE_EXE_MAX is a define that defaults to 0FFFFh. If it is nonzero it is used directly as the maximum allocation value. 0FFFFh indicates to allocate the largest memory block to the process, just like for flat .COM style load. If it is zero, then the maximum allocation field is set to equal the minimum allocation field.

Most of the stages of iniload payloads can pass through the execution control flow from being loaded as a payload to hand off to their payload. This includes the nullpl, checkpl, and inicomp stages. Each stage must be configured appropriately, enabling the _IMAGE_EXE define and setting up the other defines appropriately. These three stages expect -16:256 + 64 or -16:256 entrypoints as input for application mode, but their output entrypoints can be configured using the defines listed above.

The registers ds, es, and ax are passed through from iniload, nullpl, checkpl, or inicomp to each one's payload. The two segment registers will point to the PSP. The ax register contains information on the validity of the drives of the two default FCBs.

The iniload MZ executable header's relocation table is currently always empty. The application has to implement relocations by itself.

_SECOND_PAYLOAD_EXE allows to include a different payload that is to be used as the application mode program image. In this case the main iniload payload (_PAYLOAD_FILE) is used only for boot loaded mode. A different file can be specified with _SECOND_PAYLOAD_FILE. Each _IMAGE_EXE_* define has an equivalent define named according to the _SECOND_PAYLOAD_EXE_* scheme.

1.4 Device mode protocol

The device mode load of the payload shares its program image with the application mode. The very beginning of the image must contain a device header. (The first device header, if more than one exist.)

The inicomp, nullpl, and checkpl stages expect certain offsets for the entrypoints: 18 for the strategy, and 22 for the interrupt. Each passthrough stage must be configured with the appropriate device name and device attributes contents, and with the _DEVICE define enabled. (These defines are not needed for iniload. However, _IMAGE_EXE is required.)

The inicomp stage uses the request header's break address as passed by DOS to determine how much memory is available to the depacker. This requires MS-DOS version 5 compatibility.

1.5 Second stage FAT32 protocol, FSIBOOT

The FSIBOOT protocols are used by the lDOS FAT32 boot sector loader to access its second stage. For a sector size <= 512 Bytes the FSIBOOT stage is stored in the FSINFO sector, within the large reserved area starting 4 bytes from the beginning of the FSINFO sector. (For a sector size >= 1 KiB the FSIBOOT stage is stored in the same sector as the first boot sector loader.) The first 8 bytes of the FSIBOOT stage form a signature. The signature is unique to a given protocol revision, and has to match what is expected by the boot sector loader.

The idea is for the first stage to load the FSIBOOT stage early, then do some set up of the memory layout. Control flow is then directed into the main FSIBOOT entrypoint. A table in the first loader contains some links to the first stage, such as entrypoints to the read sector function, the success handler, the error handler, the memory full handler, and a handler to which a found directory entry is passed. Additional entries in the table point to the filename to find, the directory buffer

segment to use, the minimum number of paragraphs that should be loaded, and the segment to which the kernel file should be read.

The segment pointers for end of available load memory and the FAT buffer to use are passed in registers to the main FSIBOOT entrypoint. Some variables are passed on the stack. There are three additional entrypoints into the FSIBOOT stage, which are pointed to by the last three words of the memory allocated to FSIBOOT. Two of these are used yet, one for an additional directory search and the other for querying the size of the available Low Memory Area.

The specifics listed here are applicable to the current non-experimental protocol revision as of 2024-10-11, known as FSIBOOT5. More details as well as older protocol revisions can be found in the corresponding source texts of the lDOS FAT32 loader, stored in the repo in the boot32.asm file and in its history. Future revisions of the protocol may land in the ldosboot.exp experimental repo before they're picked into the main lDOS boot repo.

Section 2: DRLoad boot protocols

The drload stage is a replacement for lDOS's iniload initial loader. It does away with most of the entrypoints supported by iniload, including the native lDOS load protocol. This allows it to drop most of the file system and disk read related code that is included in iniload.

The motivation for drload was to minimise the overhead for a compressed Enhanced DR-DOS single-file kernel, hence the name. This allows the single-file kernel to be smaller than the sum of the two files of the prior build of a double-file kernel. The lDOS fork of MS-DOS v4.01 can use drload as well.

2.1 Sector to drload protocol

The sector to drload protocol is either the FreeDOS load protocol or the original Enhanced DR-DOS load protocol (minus the need for a second file). Refer to section 3.3 and section 3.4.

Both of these protocols share the fact that the prior loader has to load the entire file of the drload stage, so that no temporary file system support is needed in the drload stage or later in the subsequent kernel stages.

That does imply that the kernel file may not carry appended data of arbitrary size, as the entire file has to fit in the space within the Low Memory Area allocated by the prior loader.

2.2 DRLoad to payload protocol

The drload to payload protocol only sets up:

- The boot sector with FAT32 EBPB or expanded FAT32-style FAT12/FAT16 BPB
- The stack at the high end of the Low Memory Area
- The Load Command Line
- The following Load Data 1 fields:
    - ldMemoryTop
    - ldLoadTop
    - ldQueryPatchValue
- The entire next stage's payload image at a paragraph boundary

This happens to be enough to run a kernel-only inicomp stage, as well as the drkernpl stage that sets up the DRBIO/MSBIO and DRDOS/MSDOS modules and runs the former. Or, in the case of recent lDOS, the lkernpl stage setting up and running the combined MSBIO + MSDOS kernel.

Section 3: Other documented boot load protocols

This chapter documents other load protocols.

3.1 ROM-BIOS to sector load protocol

This protocol consists of:

- Linear address 07C00h holds the loaded sector
- Sector is entered at 0:7C00h (commonly) or 7C0h:0 (rarely)
- DL = load unit (reportedly may be wrong)

3.1.1 ROM-BIOS load - Assumptions

- A valid stack that does not overlap the sector is set up
- Sector size may be assumed to be exactly 512 Bytes, or at least 512 Bytes
- At least 512 Bytes from the boot device have been read
- Direction Flag is clear

3.2 MBR loader to sector load protocol

This protocol mostly mimics the ROM-BIOS to sector load protocol. However, its assumptions may differ:

- Stack top is below linear 07C00h
- DS:SI (less commonly, DS:BP) -> MBR partition table entry with effective start address of the partition being loaded
- MBR loader has relocated itself to linear 00600h before loading the next sector loader
- Currently loaded boot sector corresponds to a primary partition in the MBR's partition table

3.2.1 lDOS partition info detection

The pointer in DS:SI is expected by lDOS loaders to dynamically detect hidden sectors. This detection does some rudimentary checks before updating the hidden sectors:

- DL must be >= 80h,
- The boot flag `byte [ds:si]` must be >= 80h,

- The file system type `byte [ds:si + 4]` must not be zero,

Further, the detection can be disabled at build time or patched out using instsect's `/P NONE` switch at install time.

3.2.2 Microsoft partition type detection

It appears that some Microsoft loaders expect the loaded sector and its hardcoded hidden sectors to correspond to a primary partition in the MBR's partition table. This seems to be used to read the partition table entry's partition type byte.

3.3 Sector to FreeDOS load protocol

The FreeDOS load protocol consists of:

- The entire file is loaded
- File loaded at linear address 600h, entered at 60h:0
- BL = load unit

The default load file name is ‘KERNEL.SYS’.

3.3.1 FreeDOS load - Assumptions

Some additional assumptions about FreeDOS load:

- SS:BP -> boot sector with BPB (FAT12/FAT16) or EBPB (FAT32)
- The load unit can also be found in a BPB New field of the passed boot sector. The unit is in `byte [ss:bp + 24h]` for FAT12/FAT16 and `byte [ss:bp + 40h]` for FAT32. FAT32 is determined by `word [ss:bp + 16h] == 0`.
- The hidden sectors can be found in a BPB field of the passed boot sector. The hidden sectors are in `dword [ss:bp + 1Ch]`.
- The stack is allocated some space such that it does not overlap the load file's data (with an allocation of more than 16 bytes of unused stack space)
- The stack is not close to underflowing
- The stack is allocated behind the load file's data

The original FreeDOS loaders typically allocate their boot sectors at 1FE0h:7C00h (linear 27A00h, at 158 KiB) and the stack in the same segment with an offset that's close below this address. Note that due to their memory layout, the original loaders require the load file's size rounded up to full clusters not to exceed 134 KiB when using load segment 60h. To avoid depending on a smaller cluster size and/or to allow using load segments up to 200h, a maximum file size of 128 KiB should be assumed. (The original loaders do not **check** that the file fits, rather they may crash or corrupt memory if it is too large.)

3.3.2 FreeDOS load extension - Command line

As an extension to the FreeDOS load protocol, the lDOS load protocol's LSV extension for a command line can be used in the same way, refer to section 1.1.4. If this extension is used, all

other lDOS Load Stack Variables are not used and may be uninitialised. They will be allocated space atop the stack however.

3.4 Sector to Enhanced DR-DOS load protocol

The Enhanced DR-DOS load protocol is based on the FreeDOS load protocol. Refer to section 3.3.

The native EDR-DOS boot sector loaders are forks of the FreeDOS boot sector loaders, using a different default filename, default load/execute address, and register for the load unit.

The default load file name is ‘DRBIO.SYS’.

The FreeDOS and EDR-DOS load protocols share:

- The entire file is loaded to a segment boundary
- The stack must not overlap the loaded file data, and should be behind the data
- The load unit can also be found in a BPB New field of the passed boot sector
- The hidden sectors can be found in a BPB field of the passed boot sector

They differ in the following details:

- FreeDOS:
  - File loaded at linear address 600h, entered at 60h:0
  - SS:BP -> boot sector with BPB (FAT12/FAT16) or EBPB (FAT32)
  - BL = load unit
- Enhanced DR-DOS:
  - File loaded at linear address 700h, entered at 70h:0
  - DS:BP -> boot sector with BPB (FAT12/FAT16) or EBPB (FAT32)
  - DL = load unit

If attainable, loaders should set both DL and BL to the load unit to eliminate a difference. The same is true of DS and SS for the BPB segment.

It can be assumed in the loadee that SS = DS, eliminating one more difference between the two protocols. Further, with another assumption documented for the FreeDOS load protocol, the loadee can read the load unit from the BPB New field and ignore the BL and DL registers.

3.4.1 Enhanced DR-DOS second file

Prior to ecm's 2023 December revision of EDR-DOS, the EDR-DOS load protocol implied that the DRBIO module would be loaded as the first load file and that it would then use the hidden sectors and load unit to login a file system temporarily. The DRDOS file would be scanned for in the root directory of this FAT12, FAT16, or FAT32 file system using the name ‘DRDOS.SYS’ (generally). When found it would be loaded as the DRDOS module. In the 2023 December update, no additional DOS file is read by the single-file kernel.

3.5 Sector to MS-DOS v6 load protocol

This load protocol consists of:

- First 3 sectors of the load file are loaded
- File start is loaded at linear address 700h, entered at 70h:0
- DL = load unit
- AX:BX = first data sector of first cluster, including hidden sectors
- CH = media ID byte (albeit this is passed through msload to msbio it appears not to be used at all)
- Linear 7C00h holds boot sector with BPB
- Diskette Parameter Table (DPT) may be relocated, possibly modified. The DPT is pointed to by the interrupt 1Eh vector. Possible addresses include linear 522h and linear 7C3Eh.
- Either of the two ways of passing the original DPT address:
  - `dword [ss:sp] = 0000_0078h` (far pointer to IVT 1Eh entry), then `dword [ss:sp + 4]` -> original DPT
  - DS:SI -> original DPT
- Directory entry for first load file at linear 500h. First cluster of the file in `word [51Ah]`.
- Directory entry for second load file at linear 520h. First cluster of the file in `word [53Ah]`.

The default first load file name is ‘IO.SYS’ and the default second load file name is ‘MSDOS.SYS’.

3.5.1 MS-DOS v6 load - Assumptions

Some additional assumptions about MS-DOS v6 load:

- First sector of root directory may be read to linear 500h
- First load file may be expected in first root directory entry
- Second load file may be expected in second root directory entry
- Sector size of at least 512 Bytes
- First 1536 Bytes of first load file may be expected to be contiguous in the file system data area
- The load unit can also be found in a BPB New field of the passed boot sector. The unit is in `byte [ss:bp + 24h]` for FAT12/FAT16.
- The hidden sectors can be found in a BPB field of the passed boot sector. The hidden sectors are in `dword [ss:bp + 1Ch]`.
- The stack is allocated some space such that it does not overlap the load file's data (with an

allocation of more than 16 bytes of unused stack space)

- The stack is not close to underflowing
- The stack is allocated behind the load file's data
- The stack is allocated somewhere below linear 7C00h (between load file data and BPB)

The stack assumptions are used by lDOS iniload to distinguish MS-DOS v6 load and Enhanced DR-DOS load, as both use the same entrypoint address.

Note that BP may be uninitialised for MS-DOS v6 load.

MS-DOS v6 load is typically used only on FAT12 and FAT16 file systems,

3.6 Sector to IBM-DOS load protocol

This protocol is almost the exact same as the MS-DOS v6 protocol. However, it can be used to load from FAT32 file systems, as implemented by PC-DOS v7.10. The differences due to this are:

- Directory entry for first load file at linear 500h. First cluster low of the file in `word [51Ah]`. First cluster high of the file in `word [514h]`.
- Directory entry for second load file at linear 520h. First cluster low of the file in `word [53Ah]`. First cluster high of the file in `word [534h]`.
- The load unit can also be found in a BPB New field of the passed boot sector. The unit is in `byte [ss:bp + 24h]` for FAT12/FAT16 and `byte [ss:bp + 40h]` for FAT32. FAT32 is determined by `word [ss:bp + 16h] == 0`.

The default first load file name is ‘IBMBIO.COM’ and the default second load file name is ‘IBMDOS.COM’.

3.7 Sector to MS-DOS v7 load protocol

This load protocol consists of:

- First 4 sectors of the load file are loaded
- File start is loaded at linear address 700h, entered at 70h:200h
- DL = load unit
- `dword [ss:bp - 4]` = first data sector of first cluster, including hidden sectors
- SS:BP -> boot sector with (E)BPB, typically at linear 7C00h
- DI = first cluster of load file if FAT12/FAT16
- SI:DI = first cluster of load file if FAT32
- Diskette Parameter Table (DPT) may be relocated, possibly modified. The DPT is pointed to by the interrupt 1Eh vector.
- `dword [ss:sp] = 0000_0078h` (far pointer to IVT 1Eh entry), then `dword [ss:sp + 4]` -> original DPT

- First two bytes of load file may be expected to hold "MZ" signature
- Two bytes at offset 512 in load file may be expected to hold "BJ" signature
- `word [ss:bp + 1EEh]` points to a message table. The format of this table is described in lDebug's source files msg.asm and boot.asm, around uses of the msdos7_message_table variable.
- `byte [ss:bp + 2]` is expected to equal 0Eh or 0Ch to indicate to msload to use LBA access to the disk. These two values correspond to the FAT16 LBA or FAT32 LBA partition types as supported in the partition tables. Any other value (typically '`NOP`' 90h) indicates to use CHS access.

The default load file name is 'IO.SYS', although the names 'JO.SYS' and 'WINBOOT.SYS' may have been used in some circumstances.

3.7.1 MS-DOS v7 load - Assumptions

Some additional assumptions about MS-DOS v7 load:

- Sector size of at least 512 Bytes
- First 2048 Bytes of load file may be expected to be contiguous in the file system data area
- The load unit can also be found in a BPB New field of the passed boot sector. The unit is in `byte [ss:bp + 24h]` for FAT12/FAT16 and `byte [ss:bp + 40h]` for FAT32. FAT32 is determined by `word [ss:bp + 16h] == 0`.
- The hidden sectors can be found in a BPB field of the passed boot sector. The hidden sectors are in `dword [ss:bp + 1Ch]`.
- The stack is allocated some space such that it does not overlap the load file's data (with an allocation of more than 16 bytes of unused stack space)
- The stack is not close to underflowing

3.7.2 Multiboot v1 and Multiboot v2 specification

The load protocols for Multiboot are described by the specifications. However, iniload will drop the machine back into Real 86 Mode to run its payload. This is likely not an expected use case.

Section 4: lDOS iniload modes

The lDOS iniload stage can be loaded as the (first) load file for any of the sector load protocols described in this manual:

- lDOS / RxDOS.3
- FreeDOS
- Enhanced DR-DOS (requires file to be >= 32 KiB)
- MS-DOS v6 / IBM-DOS
- MS-DOS v7
- Multiboot v1
- Multiboot v2

After entry and loading the remainder of its payload, iniload will pass control to its payload using the iniload to payload protocol, refer to section 1.2.

The lDOS drload stage can only be loaded using the FreeDOS or EDR-DOS load protocols. As described in section 2.2 its control flow will hand over to the next stage with a subset of the iniload to payload protocol.

Source Control Revision ID

hg 2237d683f05a, from commit on at 2026-11-14 20:29:26 +0100

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/ldosboot/rev/2237d683f05a