lDebug manual

This document has been compiled on 2021-02-15.

Contents

Section 1: Overview and highlights

lDebug is a 86-DOS debugger based on the MS-DOS Debug clone FreeDOS Debug. It features DPMI client support for 32-bit and 16-bit segments, a 686-level assembler and disassembler, an expression evaluator, an InDOS and a bootloaded mode, script file reading, serial port I/O, permanent breakpoints, conditional tracing, buffered tracing, and auto-repetition of some commands. There is also a symbolic debugging branch being developed.

Section 2: Building the debugger

Building lDebug is not supported on conventional DOS-like systems. (DJGPP environments may suffice but are not tested.)

2.1 Components for building

The following components are required to build with the provided scripts:

- bash - to run mak* scripts
- perl - to patch binaries (overwrite unused revision IDs)
- grep - to detect whether boot loading is in use, and to export variables
- sed - to filter dosemu2 output
- hg (Mercurial) - to retrieve revision IDs
- python - to run hg
- C compiler - to compile supporting programs
- dosemu2 - to run build decompression tests (optional)
- qemu - to run build decompression tests (optional)
- nasm - to assemble. NASM versions to choose:
  - NASM versions up to 2.07 fail -- '`%deftok`' is not supported
  - NASM versions prior to 2.09.02 fail -- '`%deftok`' is implemented wrongly
  - NASM version 2.09.02 works (last tested 2019-11)
  - NASM versions 2.09.03 to 2.09.10 all fail -- '`%assign %$foo%[bar] quux`' doesn't function right
  - NASM version 2.10.09 works (last tested 2019-11)
  - NASM version 2.14.03 works (last tested 2020-12)
  - NASM version 2.15.03 works (last tested 2020-12)
  - NASM version 2.16 (current git head) fails, due to a bug with %strcat and a bug with %assign ?%1
- halibut - to build this manual
- supporting programs:

    - mktables (included in debugger source)
    - tellsize (included in separate repo called tellsize)
    - crc16-t/iniload/checksum (included in separate repo called crc16-t, to add checksumming, optional)
    - a 86-DOS kernel and shell (to run build decompression tests, optional)
- additional sources (must be referenced in cfg.sh or ovr.sh):
    - lmacros (macro collection)
    - scanptab (partition table scanning for bootable debugger)
    - ldosboot (iniload frame for bootable debugger, boot sector loaders)
    - instsect (application to install boot sector loaders)
    - bootimg (to run decompression test with qemu)
    - inicomp (if to use compression support), also needs one of:
        - brieflz (blzpack)
        - lz4 (lz4c)
        - snappy (snzip)
        - exomizer -- recommended as this usually results in the smallest files
        - x-compressor
        - heatshrink
        - lzip -- usually even smaller than Exomizer but takes longer to decompress
        - lzop
    - crc16-t/iniload (if to add checksumming)
    - symsnip (only for symbolic branch)

2.2 How to build

1. Clone the mercurial repo from https://hg.ulukai.org/ecm/ldebug or in an existing repo use '`hg pull`' to update the repo
2. Update the repo to either the default branch with '`hg up default`' or the symbolic branch with '`hg up symbolic`' or any other available commit you want to build
3. Clone the other needed repos from https://hg.ulukai.org/ecm/ or in existing repos use '`hg fetch`' or the sequence of '`hg pull`' then '`hg up`' to update the repos. (Usually the additional source repos do not have multiple branches.)
4. Copy the ldebug/source/cfg.sh file to ovr.sh in the same directory
5. Edit ovr.sh to point to the repos

6. Edit `INICOMP_METHOD` in ovr.sh to select none, one, or several compression methods. Surround multiple values with quotes and delimit with blanks. If the value "none" is used no compression will occur. If several values are given the smallest of the resulting files will be used as the `ldebug.com` result. This favours LZMA-lzip (lzd) and Exomizer 3 (exodecr) compression as they result in the best ratios. The uncompressed `ldebugu.com` file will always be generated, you can rename or copy or symlink it to use it as `ldebug.com` if you want.

7. If you have dosemu2 or qemu, you may enable the `use_build_decomp_test` option. This insures that the compressed executables will actually succeed in decompression when entered in EXE mode, and will lower the required minimum allocation given in the EXE header to the minimally required value so that decompression will still succeed. This defaults to using dosemu2, which must have a DOS installed that allows filesystem redirection. `DEFAULT_MACHINE` can be used to select qemu instead. The options `BOOT_KERNEL`, `BOOT_COMMAND`, and `BOOT_PROTOCOL` must be set up then to allow building a bootable diskette. (This is needed because qemu does not offer filesystem redirection for DOS.)

8. The `use_build_revision_id` option is by default on. It requires that the sources are in hg (Mercurial) repos and that the hg command is available to run '`hg id`'. The resulting revision IDs are embedded into the executable and will be shown for the ?B (long) and ?BUILD (short) commands.

9. In ovr.sh you can also specify which tools to use. For example, the variable `$NASM` specifies the nasm executable to use, with path if needed.

10. If you want to rebuild debugtbl.inc you should compile mktables then run it. While in the ldebug/source directory, run '`./makec`' (or use whatever C compiler to build mktables) then '`./mktables`' next. Note that mktables only needs to be used if either the source files (instr.*) changed or the mktables program itself has been altered. If the assembler and disassembler tables are not to change then mktables need not be used.

11. Finally, run '`./mak.sh`' from the ldebug/source directory. You may pass environment variables to it, such as '`INICOMP_METHOD=exodecr ./mak.sh`' to select Exomizer compression. You may also pass it parameters which will be passed to the main assembly command, such as '`./mak.sh -D_DEBUG4`' to enable debugging messages.

The mak.sh script expects that the current working directory is equal to the directory that it resides in. So you'll always want to run it as '`./mak.sh`' from that directory. The same is true of the make* scripts.

The make* scripts work as follows:

make

calls mak.sh to create debug and debugx

maked

calls mak.sh to create ddebug and ddebugx

maker

calls mak.sh to create only debug

makerd

calls mak.sh to create only ddebug

makex

calls mak.sh to create only debugx

makexd

calls mak.sh to create only ddebugx

ldebug/tmp, ldebug/lst, and ldebug/bin will receive the files created by the mak script. The following filenames are for the default when running mak.sh on its own which is to create debug. (When ddebug, debugx, or ddebugx are created, the names change accordingly.) In the ldebug/bin subdirectory, `debug.com` will be a nonbootable executable (even if the `_BOOTLDR` option is enabled). This executable can safely be compressed using EXE packers such as the UPX. (In cfg.sh the option `use_build_shim` now controls whether `debug.com` is created. It defaults to disable this output file.) If the `_BOOTLDR` option is enabled, `ldebug.com` will be a compressed bootable executable (if any compression method is selected), whereas `ldebugu.com` will be an uncompressed bootable executable. These bootable executables must not be compressed using any other programs. Doing that would render the kernel mode entrypoints unusable. Incidentally, UPX rejects these files because their 'last page size' MZ EXE header field holds an invalid value.

The bootable executables can be used as MS-DOS 6 protocol `IO.SYS`, MS-DOS 7/8 `IO.SYS`, PC-DOS 6/7 `IBMBIO.COM`, FreeDOS `KERNEL.SYS`, RxDOS.3 `RXDOS.COM`, or as a Multiboot specification or Multiboot2 specification kernel. In any kernel load protocol case, the root FS that is being loaded from should be a valid FAT12, FAT16, or FAT32 file system on an unpartitioned (super)floppy diskette (unit number up to 127) or MBR-partitioned hard disk (unit number above 127). In addition, the bootable executables also are valid 86-DOS application programs that can be loaded in EXE mode. (Internally, all the .com files are MZ executables with a header, but they are named with a .COM file name extension for compatibility.)

It is valid to append additional data, such as a .ZIP archive, to any of the executables. However, if too large this may render loading with the FreeDOS load protocol impossible. All the other protocols work even in the presence of arbitrarily large appended data.

2.2.1 How to build the instsect application

1. Clone the mercurial repo from https://hg.ulukai.org/ecm/ldebug or in an existing repo use '`hg pull`' to update the repo
2. Update the repo to either the default branch with '`hg up default`' or the symbolic branch with '`hg up symbolic`' or any other available commit you want to build
3. Clone the other needed repos (lmacros, ldosboot, instsect) from https://hg.ulukai.org/ecm/ or in existing repos use '`hg fetch`' or the sequence of '`hg pull`' then '`hg up`' to update the repos. (Usually the additional source repos do not have multiple branches.)
4. Copy the ldebug/source/cfg.sh file to ovr.sh in the same directory
5. Edit ovr.sh to point to the repos
6. In ovr.sh you can also specify which tools to use. For example, the variable `$NASM` specifies

the nasm executable to use, with path if needed.

7. Finally, run '`./makinst.sh`' from the ldebug/source directory. You may pass environment variables to it. You may also pass it parameters which will be passed to the assembly commands.

The makinst.sh script expects that the current working directory is equal to the directory that it resides in. So you'll always want to run it as '`./makinst.sh`' from that directory.

ldebug/tmp, ldebug/lst, and ldebug/bin will receive the files created by the makinst script. ldebug/bin/instsect.com will be the instsect application, which has boot sector loaders for FAT12, FAT16, and FAT32 embedded. The default protocol is lDOS and the default kernel name LDEBUG.COM. Read the instsect help page for instructions on how to use it. The help can be obtained by running `instsect.com /?` from DOS. The kernel name can be modified with the`/F=` switch to instsect. For instance, '`instsect.com /f=lddebugu.com a:`' installs the loader onto drive A: with the name set up to load the uncompressed lDDebug.

Current lDOS boot32 uses the FSIBOOT4 protocol for an additional stage. This is interoperable with the upcoming RxDOS version 7.25's use of the FSIBOOT4 protocol, as well as with loaders that use a different sector for their additional stage (like Microsoft's), or those that do not use an additional stage (like FreeDOS's).

2.3 Build options

_DEBUG

Make the program debuggable. A 'D' is usually prepended to the program name. This means that the program's handlers are only installed within the function run, and are uninstalled within the function intrtn1_code. This allows debugging everything except this section. This is intended to be used with a default build of lDebug as the outer debugger. However, there is nothing preventing usage of a different debugger. To indicate that the debuggable debugger is running, its default command prompts are prepended by a tilde '~'.

(To debug everything including the section from run to intrtn1_code, or the DPMI entry of lDebugX, a lower-level debugger must be used, such as dosemu's dosdebug or other debuggers that are integrated into emulators.)

_PM

Make the program DPMI-capable. An 'X' is usually appended to the program name. If possible, the interrupt 2Fh function 1687h is hooked and made to return lDebugX's entrypoint. Otherwise, the initial entry into protected mode must be traced. Upon entry lDebugX will install itself as if it is the actual client, initialise itself, then set up the original client as if that had entered protected mode. The assembler and disassembler will detect and support 32-bit code segments. Other commands will also use 32-bit addressing to allow using 32-bit segments. To indicate that the debugger is in protected mode, its default command prompt changes from the dash '`-`' to a hash sign '#'. (lDDebugX prepends its tilde to that resulting in '~#'.)

_BOOTLDR

Makes the program support being bootloaded. This additionally requires the lDOS iniload stage wrapped around the MZ .EXE image of the debugger. The mak.sh script prepends

an ‘l’ to the base filename to create the names for the bootable files. For building debug, this results in `ldebugu.com` and `ldebug.com`. In bootloaded mode, I/O is never done using DOS, as if InDOS mode was always on. The DOS's current PSP is not switched during debugger operation. The MCB chain can only be displayed using the DM command by specifying the start segment explicitly. The BOOT commands are supported, refer to section 7.10.

Section 3: Invoking the debugger

3.1 Invoking the debugger in boot loaded mode

The debugger can be loaded as a variety of kernel formats.

The Multiboot1 and Multiboot2 entrypoints will expect that a kernel command line is provided. The RxDOS.3 and lDOS load protocols allow specifying a kernel command line, but it is optional.

If a kernel command line is detected then its contents are entered into the command line buffer. Unescaped semicolons are translated into Carriage Returns. Semicolons and backslashes may be escaped with backslashes.

If no kernel command line is given, the debugger assumes a default. It is equivalent to checking for a file and label using the IF command (section 5.18), then if found to execute that script file. The IF condition is like `if exists y ldp/LDEBUG.SLD :bootstartup then` and the subsequent script command is `y ldp/LDEBUG.SLD :bootstartup` (section 5.39). The filename is however `LDDEBUG.SLD` for DDebug builds.

Executing the `Q` command (section 5.26) makes the debugger uninstall itself then continue running whatever code the debuggee is in. Executing the `BOOT QUIT` command (section 7.10) makes the debugger attempt to shut down the machine. First it will try to call a dosemu-specific callback. Next it will attempt shutting down with APM. (This works in qemu.) Finally it will give up if no attempt worked.

3.2 Invoking the debugger as an application

The debugger is internally an MZ .EXE style application. It may need MS-DOS version 3 level features. A few switches are supported:

`/?`

Show the online help page about invoking the debugger.

`/c`

Put the text following this switch into the command line buffer. Unquoted unescaped blanks indicate the end of the text. Parts may be quoted using single quote marks or double quote marks. Unescaped semicolons are translated into Carriage Returns. Semicolons, backslashes, quote marks, and blanks may be escaped with backslashes.

`/z`

This switch is only used by the symbolic branch. It can be used to set the size of the symbol tables early, before loading a debuggee application.

After the switches a filename may follow. After the filename, command line contents for the process to be debugged may follow. These are both passed to the N command. Then, an L command for loading an application is run.

Executing the Q command (section 5.26) makes the debugger try to terminate the debuggee application and to then terminate itself. The debugger returns to whatever application called it.

If the TSR command (section 5.34) is used, the debugger patches the parent of the currently running application to be the debugger's parent. A subsequent Q command will then behave much like it does in boot loaded mode: The debugger uninstalls itself and continues execution in the current debuggee context.

Section 4: Parameter Reference

4.1 Number

Plain numbers are evaluated as expressions. Expressions consist of any number of the following:

- Unary operators
- Binary operators
- Operands

Plain number parsing for an expression continues for as long as a valid expression is continued. For example, in the command '`D 100 + 20 L 10`' the starting address (its offset to be specific) is calculated as '100 + 20'. Then the expression evaluator encounters the '`L`', which is not a valid binary operator. Plain number expression parameters are used by a lot of commands. Sometimes, the plain number parameter type is called 'count' or 'value'.

4.2 Address

An address parameter is calculated with a default segment. First, a plain number is parsed. If it is followed by a colon, the first number is taken as segment, and then another number is parsed for the offset. Otherwise, the first number is used as the offset. Offsets may be 16 bits or 32 bits wide, though 32-bit offsets are only valid for DebugX and only in 32-bit segments. Address parameters are used by a lot of commands.

4.3 Range

A range parameter may have a default length, or it may be disallowed to omit a length. Parsing a range starts with parsing an address. Then, if the end of the line is not yet reached, an end for the range may be specified. The end may be a plain number, which is taken as the offset of the last byte to include in the range. The address of the last byte to include must be equal or above the address of the first byte that is included in the range.

The end may instead be specified with an '`L`' or '`LENGTH`' keyword. In that case, the keyword is followed by a plain number and an optional item size keyword. A length of zero is not valid. The item size keyword may be '`BYTES`', '`WORDS`', or '`DWORDS`'. For the latter two, the plain number will be multiplied by 2 or 4. The '`BYTES`' keyword is only provided for symmetry; currently all commands taking ranges default to byte size for the '`LENGTH`' number.

For example, the command '`DD 100 LENGTH 4 DWORDS`' will dump memory from address 0100h (in the current data segment) in dword units, for a length of 4*4 = 16 bytes. The item size keywords were introduced primarily for the '`DW`' and '`DD`' commands (refer to section 5.8), but they can be used for any command that accepts a range.

Range parameters are used by a lot of commands.

4.4 List

A list is made up of a sequence of items. Each item is either a plain number or a quoted string. List parsing continues until the end of the line. Each plain number represents a single byte. Quoted strings represent as many bytes as there are quoted. List parameters are used by the E, F, and S commands. Refer to section 5.12, section 5.13, and section 5.31.

4.5 List or range

A list or range can be specified for this parameter. The range is identified by a leading ‘RANGE’ keyword. Otherwise, a list is parsed. A list or range parameter is as yet used by the S command and the F command, refer to section 5.31 and section 5.13.

4.6 Keyword

A keyword is checked insensitive to capitalisation. Keywords depend on each command. Only the keywords used to specify a range's length are shared by all commands that parse ranges.

4.7 Index

An index is a plain number that specifies a breakpoint index. It allows operating on one specific breakpoint. The index parameter type is used by the B commands, refer to section 5.5.

4.8 Segment

A segment is a plain number for parsing purposes. The segment parameter type is used by the DM command and some BOOT commands, refer to section 5.10 and section 7.10.

4.9 Breakpoint

Each breakpoint is a single address, which defaults to the code segment. The address may instead be specified starting with an AT sign ‘@’, followed by a blank or an opening parenthesis. In that case, the following plain number specifies the non-segmented linear address to use. The breakpoint parameter type is used by the B and G commands, refer to section 5.5 and section 5.14.

4.10 Label

A label is a (not quoted) string keyword. It may start with an optional colon. A label can be used by the GOTO and Y commands, refer to section 5.15 and section 5.39.

4.11 Port

A port is a plain number for parsing purposes. The port parameter type is used by the I and O commands, refer to section 5.17 and section 5.24.

4.12 Drive

A drive may be either an alphabetic letter followed by a colon, or a plain number. The number zero corresponds to drive A: then. The drive parameter type is used by the L and W sector commands, refer to section 5.20 and section 5.37. The N and Y commands (section 5.23 and section 5.39) also accept drive parameters, but only as part of their filenames. These must be in the drive letter followed by colon format.

4.13 Sector

A sector is a plain number, which can be equal to any 32-bit value. The sector parameter type is used by the L and W sector commands, refer to section 5.20 and section 5.37. Some BOOT commands also use sector numbers, refer to section 7.10.

4.14 Condition

A condition is a plain number. It is evaluated either to nonzero (true) or zero (false). The condition parameter type is used by the IF command, as well as the P, TP, and T commands when specified with a '`WHILE`' keyword. The BW and BP (with a '`WHEN`' keyword) commands also use conditions. Refer to section 5.18, section 5.25, section 5.32, section 5.5.3, section 5.5.1. The length of a condition for B commands is limited by how much space is left in the permanent breakpoint conditions buffer. This buffer currently defaults to 1024 bytes. It is shared for all conditions of all permanent breakpoints.

4.15 Register

A register specifies an internal variable of the debugger. Most prominently these include the debuggee's registers as stored by the debugger in its data segment. A register or variable may be an operand in a plain number's expression. However, several forms of the R command also use register parameters. These allow reading and writing the register values. Refer to section 5.27.

4.16 Command

Command is a special parameter type that is used only by the RE.APPEND and RE.REPLACE commands (section 5.27.2). It is read verbatim and entered into the RE command buffer. Semicolons within a command parameter are not parsed as end of line comment markers. Instead, they are converted to CR (13) codes in the RE buffer. This delimits the parts of the parameter into several commands. A semicolon may be prefixed by a backslash to escape it and thus enter a literal semicolon into the RE buffer.

4.17 ID

ID is a special parameter type that is used only by the BP and BI commands (section 5.5.1 and section 5.5.2). Leading and trailing whitespace is ignored. An ID can be empty, or contain up to 63 bytes of data. The length of an ID is also limited by how much space is left in the permanent breakpoint ID buffer. This buffer currently defaults to 384 bytes. It is shared for all IDs of all permanent breakpoints.

Section 5: Command Reference

5.1 Empty command - Autorepeat

Entering an empty command at an interactive prompt results in autorepeat. Interactive prompts for this purpose include:

- the debugger as a DOS application (`int 21h`)
- the debugger in InDOS mode or as a bootloaded program (`int 16h`/`int 10h`)
- the debugger across a serial port (port I/O)

Input that does not count as an interactive prompt includes:

- reading from a file redirected as stdin using DOS (`int 21h`)
- reading from a Y script file using DOS (`int 21h`)
- reading from a Y script file while bootloaded (`int 13h`)
- reading from the command line buffer
- reading from the RE buffer

Autorepeat is not supported by all commands. The following commands support autorepeat:

D/DB/DW/DD

> Continues memory dump behind the last prior dumped position. Continues with the same size as the prior dump. As for if the command is executed with an address lacking a length, the default length (128 bytes) is used.

DZ/D$/D#/DW#

> Continues string dump behind the last prior dumped string. Continues with the same type of string as the prior dump.

DX

> Continues memory dump.

G

> Repeats a step running the debuggee. An equals address given to the prior Go command is not used again. The same G breakpoints as used by the prior Go command are used (same as G AGAIN). The exception is that wherever a breakpoint matches the `CS:(E)IP` at the start of the command's execution, it is skipped once.

P

Repeats a step running the debuggee. An equals address given to the prior Proceed command is not used again. A count given to the prior Proceed command is not used again, autorepeat always runs as if not given a count. (That means the PPC variable is used as the effective count. Refer to section 6.3.)

T

Repeats a step running the debuggee. An equals address given to the prior Trace command is not used again. A count given to the prior Trace command is not used again, autorepeat always runs as if not given a count. (That means the TTC variable is used as the effective count. Refer to section 6.3.)

TP

Repeats a step running the debuggee. An equals address given to the prior Trace/Proceed command is not used again. A count given to the prior Trace/Proceed command is not used again, autorepeat always runs as if not given a count. (That means the TPC variable is used as the effective count. Refer to section 6.3.)

U

Repeats disassembly behind the last prior disassembled instruction. As for if the command is executed with an address lacking a length, the default length (32 bytes) is used.

5.2 ? command

```
Online help    ?
```

The question mark command (?) lists the main online help screen.

There are additional help topics that can be listed by using the question mark command with an additional letter or keyword. These keywords are as follows:

```
Registers      ?R
Flags          ?F
Conditionals   ?C
Expressions    ?E
Variables      ?V
R Extended     ?RE
Run keywords   ?RUN
Options        ?O
Boot loading   ?BOOT
lDebug build   ?BUILD
lDebug build   ?B
lDebug sources ?SOURCE
lDebug license ?L
```

The full help pages are listed in section 7.

5.3 : prefix - GOTO label

A leading colon indicates a destination label for GOTO, see section 5.15.

5.4 A command - Assemble

```
assemble        A [address]
```

Starts assembly at the indicated address (which defaults to CS segment), or if no address is specified, at the "a_addr" (AAS:AAO variables).

Assembly mode has its own prompt. Entering a single dot (.) or an empty line terminates assembly mode. Comments can be given with a prefixed semicolon. In assembly mode, whereever an immediate number occurs an expression can be given surrounded by parentheses (and). In such expressions, register names like AX are evaluated to the values held by the registers at assembly time. To refer to a register as an assembly operand, it must occur outside parentheses.

5.5 B commands - Permanent breakpoints

There are a fixed number of permanent breakpoints provided by the debugger. The default is to provide 16 permanent breakpoints. They are specified by indices ranging from 00 to 0F. A breakpoint can be unused, used while enabled, or used while disabled. A breakpoint that is in use has a specific linear address. It is allowed, though not advised, for several breakpoints to be set to the same address.

When running the debuggee with the commands G, T, TP, or P, hitting a permanent breakpoint stops execution, and indicates in a message "Hit permanent breakpoint XX" where XX is replaced by the hexadecimal byte index of the breakpoint. If the breakpoint counter is not equal to 8000h when the breakpoint is hit, then the "Hit" message is followed by a "counter=YYYY" indicator. If the breakpoint ID is not empty, then the ID is shown with an "ID: " prefix. The ID is shown either on the same line as the "Hit" message, or on the next line if the ID exceeds 28 bytes. After that message a register dump occurs, same as for default breaking for the Run commands.

The exceptions are as follows:

- If the CS:(E)IP at the first step of a G command matches any breakpoints, then G does a TP-like step with all breakpoints other than the "cseip"-breakpoint written, while the "cseip"-breakpoint is not written. After that, the "cseip"-breakpoint is written and execution resumes as normal for G.
- If T.NB or TP.NB or P.NB is used, no permanent breakpoints are written at all.
- If T.SB or TP.SB or P.SB is used, then during the first step no permanent breakpoints are written. If a counter higher than 1 is given, then during subsequent steps permanent breakpoints are written.

Each breakpoint has a breakpoint counter, which defaults to 8000h if not set explicitly by the BP or BN commands. The breakpoint counter behaves as follows:

- If (counter & 3FFFh) equals zero then the counter is considered to be at a terminal state.
- If the point breaks while the counter is not at a terminal state, then the counter is decremented.

- If the counter is decremented to 0 or 4000h, then the point is hit.
- If the counter is decremented to 8000h or C000h, or was already at either count without being decremented, then the point is hit.
- If the point is not hit but the bit (counter & 4000h) is set, then the point is passed.

The point being passed means that during running the debuggee with a Run command, execution is not stopped, but a message indicating "Passed permanent breakpoint XX, counter=YYYY" is displayed. As for the "Hit" message the ID, if any, is also shown. After that message, a register dump occurs. Then execution is continued in accordance with the command that is running debuggee code.

Each breakpoint can have a breakpoint condition. If the condition expression evaluates to false when the point breaks, then the point is not considered hit or passed. The breakpoint counter is not stepped then either.

5.5.1 BP command - Set breakpoint

```
set breakpoint  BP index|AT|NEW address
                  [[NUMBER=]number] [WHEN=cond] [ID=id]
```

BP initialises the breakpoint with the given index. It must be a yet unused breakpoint. If the index is specified as the keyword NEW, the lowest unused breakpoint (if any) is selected. If there is the keyword AT instead of an index or a keyword NEW, then an existing breakpoint at the same linear address is reset, or a new one is added (same as if given the NEW keyword).

The address can be given in a segmented format, which defaults to CS, and which in DebugX is subject to either PM or 86M segmentation semantics depending on which mode the debugger is in. The address can also be given with an @ specifier (followed by an opening parenthesis or whitespace) in which case it is specified as the 32-bit linear address. Debug without DPMI support limits breakpoints to 24-bit addresses, of which 21 bits are usable.

The optional number, which defaults to 8000h, sets the breakpoint counter to that number.

The optional WHEN keyword introduces a breakpoint condition. If the breakpoint is reached then the condition, if specified, is checked before stepping the counters. If the condition is false at that point the point is not considered hit or passed and its counter is not stepped.

There is an optional OFFSET keyword (not shown in the example) which allows overriding the breakpoint's preferred offset. Refer to section 5.5.4 for details.

The optional ID keyword allows setting the breakpoint ID. The ID is displayed by BL and when a breakpoint is hit or passed. The default ID is an empty ID. Note that the ID extends for the remainder of the line. There cannot be a breakpoint counter number nor WHEN condition nor OFFSET after the ID keyword.

5.5.2 BI command - Set breakpoint ID

```
set ID          BI index|AT address [ID=]id
```

BI sets the breakpoint ID of the specified breakpoint. The ID is displayed by BL and when a breakpoint is hit or passed. The ID may be specified as empty.

5.5.3 BW command - Set breakpoint condition

```
set condition  BW index|AT address [WHEN=]cond
```

The BW command sets the breakpoint condition. If the WHEN keyword and the condition are absent then the condition is reset. That means the point is no longer conditional.

5.5.4 BO command - Set breakpoint preferred offset

```
set offset     BO index|AT address [OFFSET=]number
```

The BO command sets the breakpoint preferred offset. The preferred offset is used only by the BL command. It is used to determine the segmented address to display. The offset is a word variable for Debug and a dword variable for DebugX. If the OFFSET keyword and the number are absent then the offset is disabled, as if the breakpoint was specified with a linear address. (Internally this is done by setting the offset to all 1 bits. The offset can be explicitly set to FFFFh (Debug) or FFFF_FFFFh (DebugX) for the same effect.)

5.5.5 BN command - Set breakpoint number

```
set number     BN index|AT address|ALL number
```

BN sets the breakpoint counter of the specified breakpoint with the given index, or all used breakpoints when given the keyword ALL, or the first breakpoint with a matching linear address when given the AT keyword. The number defaults to 8000h.

5.5.6 BC command - Clear breakpoint

```
clear          BC index|AT address|ALL
```

BC clears the specified breakpoint with the given index, or all breakpoints when given the keyword ALL, or the first breakpoint with a matching linear address when given the AT keyword. This returns the specified breakpoint (or all of them) to the unused state. Any associated ID or condition is deleted by BC too.

5.5.7 BD command - Disable breakpoint

```
disable        BD index|AT address|ALL
```

Given an index or the keyword ALL or the keyword AT (like BC), BD disables breakpoints that are in use. A disabled breakpoint's address is retained and BP will not allow initialising it anew (except with AT), but it is otherwise skipped in breakpoint handling.

5.5.8 BE command - Enable breakpoint

```
enable         BE index|AT address|ALL
```

Like BD, but enables breakpoints.

5.5.9 BT command - Toggle breakpoint

```
toggle         BT index|AT address|ALL
```

Like BE and BD, but toggles breakpoints: A disabled breakpoint is enabled, while an enabled breakpoint is disabled.

5.5.10 BL command - List breakpoints

```
list            BL [index|AT address|ALL]
```

BL lists a specific breakpoint given by its index, or all used breakpoints if given the keyword ALL or given neither an index nor the keyword. When given the AT keyword, all breakpoints with a matching linear address are listed. (This differs from all other B commands, which only select the first matching breakpoint when the AT keyword is given.)

When listing all breakpoints only used breakpoints are displayed.

The output format for unused breakpoints is as follows:

- "BP"
- The byte index given as two hexadecimal digits
- "Unused"

The output format for used breakpoints is as follows:

- "BP"
- The byte index given as two hexadecimal digits
- A plus sign if the breakpoint is enabled, a minus sign if it is disabled.
- "Lin=" followed by the linear address of this breakpoint.
- The segmented address of this breakpoint. Only displayed if the breakpoint was initially specified with a segmented address, or it had a preferred offset specified with the BP OFFSET= keyword or to the BO command.
- The breakpoint content byte given in parentheses (generally "CC").
- "Counter=" followed by the breakpoint counter.
- "ID: " followed by the breakpoint ID, if any. Depending on the length the ID is shown on the first line or on a second line.
- "WHEN " followed by the breakpoint condition, if any. This is always written to a line on its own.

Example output of BL:

```
-bp at 100 id = start
-bp at 103 counter = 4000
-bp at 105 when al == 7
-bl
BP 00 + Lin=01_BB70  1BA7:0100 (CC) Counter=8000, ID: start
BP 01 + Lin=01_BB73  1BA7:0103 (CC) Counter=4000
BP 02 + Lin=01_BB75  1BA7:0105 (CC) Counter=8000
 WHEN al == 7
-
```

5.6 BU command - Break Upwards

```
break upwards   BU
```

This command, which is only supported by Debuggable lDebug builds (DDebug), causes the debugger to execute an int3 instruction in its own code segment. This breaks to the next debugger that was installed prior to DDebug. Prior to the breakpoint, the message "Breaking to next instance." is displayed.

In non-debuggable lDebug builds, the following error message is displayed instead:

```
-bu
Already in topmost instance. (This is no debugging build of lDebug.)
-
```

5.7 C command - Compare memory

```
compare         C range address
```

Given a range, the address of which defaults to DS, and another address that also defaults to DS, this command compares strings of bytes, and lists the bytes that differ.

5.8 D command - Dump memory

```
dump            D [range]
dump bytes      DB [range]
dump words      DW [range]
dump dwords     DD [range]
```

Given a range, the address of which defaults to DS, this command dumps memory in hexadecimal and as ASCII characters. If the DCO option 4 is set, characters with the high bit set (80h to FFh) are displayed as-is in the character dump. Otherwise, they will be treated like control characters, which means replaced by dots.

If no range is specified, the D command continues dumping at "d_addr" (ADS:ADO), which is updated by each D command to point after the last shown byte.

The default is for D to dump bytes. After a DW or DD command, the autorepeat and plain D (without a range) default to the last-used size. If the default range should be used but the size should be reset to bytes, the DB command can be used. The D command with a range always acts the same as DB.

5.9 DI command - Dump Interrupts

```
dump interrupts DI interrupt [count]
```

The DI command dumps interrupt vectors from the IVT (86M) or IDT (PM). In PM, for the vectors 00h to 1Fh, the exception handlers are also dumped.

5.10 DM command - Dump MCBs

```
dump MCB chain  DM [segment]
```

The DM command dumps an MCB chain. If not given a start MCB segment, and the debugger is

running as an 86-DOS application, the start of DOS's MCB chain is used. If given a start MCB segment, this is used as the starting MCB. (Note: In current RxDOS builds, the start MCB is always at segment 60h.)

The DM command initially lists the debuggee's PSP. This is only valid when the debugger is running as an 86-DOS application.

The MCB chain dump is continued until an MCB is encountered that has neither an M nor a Z signature letter, or the MCB address wraps around the 1 MiB boundary. In particular, this means that a disabled UMB link MCB (usually pointing to the MCB at segment 9FFFh if there is no EBDA nor any pre-boot-loaded programs) will not end the dump.

Example output:

```
-dm
PSP: 1A73
02B4 4D 0008 0016    352 B SD
02CB 4D 02CC 00BC    2 KiB COMMAND
0388 4D 039D 0013    304 B SYSTEM
039C 4D 039D 0034    832 B SYSTEM
03D1 4D 04A3 0013    304 B LDEBUG
03E5 4D 03E6 00BC    2 KiB COMMAND
04A2 4D 04A3 15CF   87 KiB LDEBUG
1A72 5A 1A73 858C  534 KiB DEBUGGEE
9FFF 4D 0008 3100  196 KiB SC
D100 4D 0008 1EFF  123 KiB SC
F000 4D 02CC 0040   1024 B COMMAND
F041 4D 0000 0492   18 KiB
F4D4 4D 0000 0619   24 KiB
FAEE 4D 0000 0090    2 KiB
FB7F 5A 03E6 0080   2048 B COMMAND
-
```

The columns are as follows:

1. Segment address of MCB in hexadecimal. Always one less than the segment of the memory block contents.

2. Signature letter in hexadecimal. Usually 4D (‘M’) for linking MCB and 5A (‘Z’) otherwise.

3. Owner of the MCB in hexadecimal. Values below 50h are special system values. 0 indicates an unused MCB. 8 is the usual SC/SD/S system MCB owner. Higher values are generally process segments. A process segment is usually a memory block that is preceded by an MCB, which is owned by that block itself.

4. Size in paragraphs of the MCB in hexadecimal. A value of zero is valid and indicates an MCB with an empty corresponding memory block.

5. Size in bytes or kibibytes, in decimal.

6. Name of the owner of this MCB. Free MCBs do not have a name. System MCBs have a name that is up to two letters long. Otherwise, the name is read from the MCB owner's own MCB. In this case the name is up to 8 letters long.

5.11 DZ/D$/D#/DW# commands - Dump strings

```
display strings DZ/D$/D[W]# [address]
```

The D string commands each dump a string at a specified address, which defaults to DS as the segment.

- DZ displays an ASCIZ string, terminated by a byte with the value 0.
- D$ displays a CP/M-style string, terminated by a dollar sign character $.
- D# displays a Pascal-style string with a length count in the first byte.
- DW# displays a string with a length count in the first word.

5.12 E command - Enter memory

```
enter           E address [list]
```

5.13 F command - Fill memory

```
fill            F range [RANGE range|list]
```

The F command fills memory with a byte pattern. The first parameter is the range to fill. The next parameter can be a list, in which case it provides the pattern with which to fill. If the RANGE keyword is provided then the pattern is read from memory as indicated by the range parameter that follows the keyword. The pattern is repeated so as to fill the destination. If the RANGE keyword is used, then the length of the pattern address range is optional. If the length is absent, it is assumed to equal that of the destination range.

5.14 G command - Go

```
go              G [=address] [breakpts]
```

The G command runs the debuggee. It can be given a start address (the segment of which defaults to CS), prefixed by an equals sign, in which case CS:EIP is set to that start address upon running. Note that if there is an error parsing the command line, CS:EIP is not changed. Further, if a breakpoint fails to be written initially, CS:EIP also is not changed.

The G command allows specifying breakpoints, which are either segmented addresses (86M or PM addresses depending on DebugX's mode) or linear addresses prefixed by an "@ " or "@(", similar to how the BP command allows a breakpoint specification. G breakpoints are identified by their position in the command line, as the 1st, 2nd, 3rd, etc. By default, 16 G breakpoints are supported.

The G AGAIN command re-uses the breakpoints given to the last (successfully parsed) G command. It also allows an equals-sign-prefixed start address like the plain G command, in front of the AGAIN keyword. After the AGAIN keyword, additional breakpoints may be specified.

If the command repetition of G is used, it is handled as if "G AGAIN" was entered, that is it re-uses the same breakpoints as those given to the prior G command.

A G command that fails to parse will not modify the stored G breakpoint list. If an error occurs

during writing breakpoints, the list will have been modified already however.

The G LIST command lists the breakpoints given to the last (successfully parsed) G command.

The "content" byte in G LIST is usually CCh (the int3 instruction opcode), but retains its original value if a failure occurs during breakpoint byte restoration.

Example output of G LIST:

```
-g 100 103 105
AX=3000 BX=0000 CX=0200 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=1BA7 ES=1BA7 SS=1BA7 CS=1BA7 IP=0103 NV UP EI PL ZR NA PE NC
1BA7:0103 CD21              int     21
-g list
   1st G breakpoint, linear 0001_BB70  1BA7:0100, content CC
   2nd G breakpoint, linear 0001_BB73  1BA7:0103, content CC (is at CS:IP
   3rd G breakpoint, linear 0001_BB75  1BA7:0105, content CC
-
```

The output is as follows:

- The 1-based index ordinal of the point.
- The linear address of the point. (21-bit for Debug, 32-bit for DebugX.)
- The segmented address of the point. Only listed if the point was specified in a segmented form. That is, if the point was specified with a "@ " or "@(" prefix then no segmented address is saved along with it. (Internally, the word or dword "preferred offset" variable is set to all 1 bits then.) In Protected Mode, the segment is specified as ‘`CS:`’ if the code segment's base matches the preferred offset. Otherwise, an R86M segment is shown with a dollar sign ‘`$`’ prefix if the preferred offset matches any R86M segment. Failing that the offset is shown with a prefix reading ‘`????:`’.
- The content byte. This is usually CCh. However, if a breakpoint failed to be restored then the original value is displayed here.
- Indicator that this point matches the current CS:IP or CS:EIP. This is only displayed if such a match is applicable. Running G AGAIN when this is applicable will step one time to bypass the corresponding point.

There is another G command: After any equals sign, AGAIN keyword, and/or specified breakpoints, the line can be ended with a REMEMBER keyword. This saves the specified G breakpoint list and then returns control to the user. (The equals address, if any, is discarded.) It allows preparing a G breakpoint list ahead of its use. Auto-repeat, if enabled, will run like G AGAIN and actually run the debuggee after a G REMEMBER command.

5.15 GOTO command - Control flow branch

`goto` `GOTO :label`

The GOTO command can only be used when executing from a script file, the command line buffer, or the RE buffer. It lets execution continue at a different point in the file or buffer. Labels are identified by lines that start with a colon, followed by the alphanumeric label name, and

optionally followed by a trailing colon. The destination label of the GOTO command may be specified with or without the leading colon.

There are several special cases:

- If the destination label is :SOF (Start Of File) then the file or buffer completely rewinds to its start.
- If the destination label is :EOF (End Of File) then the file or buffer is closed.
- If the destination label is not found then the file or buffer is closed, along with an error message.

5.16 H command - Hexadecimal add/subtract values

```
hex add/sub      H value1 [value2 [...]]
```

The H command performs calculation and displays the result. If a single expression is given then its value is displayed, in hexadecimal and then in decimal. If more than one expression is given then two results are displayed, in hexadecimal only. The first result is that which is calculated by adding all expressions. The second result is calculated by subtracting all subsequent expressions from the first expression's value.

If a value is above or equal to 8000_0000h then along each display of that value, the value interpreted as a negative two's complement number is listed in parentheses.

Examples:

```
-h 1
0001  decimal: 1
-h 1 1
0002  0000
-h 1 1 1
0003  FFFFFFFF (-0001)
-h 1 + 2 * 3
0007  decimal: 7
-h cs * 10
0001A730  decimal: 108336
-h -26
FFFFFFDA (-0026)  decimal: 4294967258 (-38)
-
```

5.17 I command - Input from port

```
input            I[W|D] port
```

The I commands input from an x86 port. The port can be any number between 0 and FFFFh. Plain I inputs a byte from the specified port. The IW and ID commands input a word or dword respectively.

5.18 IF command - Control flow conditional

```
if numeric       IF [NOT] (cond) THEN cmd
```

```
if script file  IF [NOT] EXISTS Y file [:label] THEN cmd
```

The IF command allows specifying a conditionally executed command. This is especially useful for creating conditional control flow branches with the GOTO command (see section 5.15).

For the first form, the condition is a numeric expression. If it evaluates to non-zero it is considered true. If the NOT keyword is absent then a true condition expression leads to executing the THEN command. With the NOT keyword present the logic is reversed. Note that if an error occurs in parsing, the THEN command is not executed, regardless of whether the NOT keyword is present.

The second form specifies a script file in the same format as accepted by the Y command (refer to section 5.39). A label may be specified behind the filename, as for the Y command. If the file is found, and contains the specified label if any, then the EXISTS clause is considered true. Depending on the presence of the NOT keyword the THEN command is executed next, or skipped. Note that if an error occurs in parsing, the THEN command is not executed, regardless of whether the NOT keyword is present.

Likewise, if an unanticipated error occurs during access then the THEN command is not executed. Anticipated errors include:

1. The drive or ROM-BIOS unit cannot be accessed at all. (Determined by sector 0 being unreadable.)
2. The specified partition is not found.
3. A specified directory is not found.
4. The file is not found.
5. A DOS error occurs opening the file.
6. The file is empty.
7. A specified label is not found.

5.19 L command - Load Program

```
load program    L [address]
```

5.20 L command - Load Sectors

```
load sectors    L address drive sector count
```

5.21 M command - Move memory

```
move            M range address
```

5.22 M command - Set Machine mode

```
80x86/x87 mode  M [0..6|C|NC|C2|?]
```

5.23 N command - Set program Name

```
set name        N [[drive:][path]progname.ext [parameters]]
```

5.24 O command - Output to port

```
output          O[W|D] port value
```

The O commands output to an x86 port. The port can be any number between 0 and FFFFh. Plain O outputs a byte to the specified port. The OW and OD commands output a word or dword respectively. The value to write is specified by the second expression.

5.25 P command - Proceed

```
proceed         P [=address] [count [WHILE cond] [SILENT [count]]]
```

The P command causes debuggee to run a proceed step. This is the same as tracing (T command) for most instructions, but behaves differently for "call", "loop", and repeated string instructions. For these, a proceed breakpoint is written behind the instruction (similarly to how the G command writes breakpoints), and the debuggee is run without the Trace Flag set.

Like for the G command, a start address can be given to P prefixed by an equals sign. Next, a count may be specified, which causes the command to execute as many P steps as the count indicates.

After a count, a WHILE keyword may be specified, which must be followed by a conditional expression. Execution will only continue if the WHILE expression evaluates to true.

After a count (when no WHILE is given) or after a WHILE condition, a SILENT keyword and optional count may be given. In this case, the debugger buffers the register dump and disassembly output of the executed steps, until control returns to the debugger command line. Then, the last dumps stored in the buffer are displayed. If a non-zero count is given, at most that many register dumps are displayed.

5.26 Q command - Quit

```
quit            Q
```

5.27 R command - Display and set Register values

```
register        R [register [value]]
```

The R command without any register specified dumps the current registers, either displayed as 16-bit or 32-bit values (depending on the RX option), and disassembles the instruction at the current CS:(E)IP location.

R with a register, named debugger variable, or memory variable (of the form BYTE/WORD/3BYTE/DWORD [segment:offset]) displays the current value of the specified variable. It then displays a prompt, allowing the user to enter a new value for that variable. Entering a dot (.) or an empty line returns to the default debugger command line.

R with a variable, followed by a dot (.), only displays the current value of that variable.

R with a variable, followed by an optional equals sign, and followed by an expression, evaluates the expression and assigns its resulting value to the variable. The equals sign may instead be a binary operator with a trailing equals sign, which is handled as an assignment operator.

Examples:

```
-r ax .
AX 0000
-r ax
AX 0000  :1
-r ax
AX 0001  :.
-r ax += 4
-r ax
AX 0005  :
-r word [cs:0]
WORD [1867:0000] 20CD  :
-r dif .
DIF 0100B00B
-
```

5.27.1 RE command - Register dump Extended

```
R extended      RE
```

The RE command runs the RE buffer commands. Refer to section 7.7.

5.27.2 RE buffer commands

```
RE commands     RE.LIST|APPEND|REPLACE [commands]
```

RE.LIST lists the RE buffer contents in a way that can be re-used as input to RE.REPLACE.

RE.APPEND appends the following commands to the RE buffer.

RE.REPLACE replaces the RE buffer with the following commands.

The RE buffer usage is described in the ?RE help page (section 7.7).

5.28 RM command - Display MMX Registers

```
MMX register    RM
```

5.29 RN command - Display FPU Registers

```
FPU register    RN
```

5.30 RX command - Toggle 386 Register Extensions display

```
toggle 386 regs RX
```

5.31 S command - Search memory

```
search          S range [REVERSE] [RANGE range|list]
```

The S command searches memory for a byte string. The first range specifies the search space. By default, searching will begin at the bottom of the search space and move upwards. If a REVERSE keyword is specified after the range then searching will begin at the top of the search space moving downwards. The search string is specified either with the RANGE keyword followed by another range, or as a list of byte values.

The read-only variable SRC (Search Result Count) will receive the 32-bit value that is the amount of matched occurrences. The variable SRS0 receives the first Search Result Segment. Likewise SRO0 receives the first Search Result Offset. SRO1 to SROF hold subsequent Search Result Offsets. SRO is an alias to SRO0. SRO variables are 32-bit in the _PM build lDebugX, 16-bit otherwise. Unused SRO variables are zeroed out by a successful search.

The display of search results is as follows:

- First, the result's segmented address.
- Then, a hexadeximal dump of the 16 bytes that follow the search string match at this point.
- Finally, the ASCII character dump of these 16 bytes.

There is an option to disable the data dump so as to only display the match addresses. If the bit 80_0000h is set in the DCO variable then the data dump is suppressed.

5.32 T command - Trace

```
trace           T [=address] [count [WHILE cond] [SILENT [count]]]
```

The T command is similar to the P command. However, T traces most instructions. Depending on the TM option, interrupt instructions are also traced (into the interrupt handler) or proceeded past.

5.32.1 TP command - Trace/Proceed past string ops

```
trace (exc str) TP [=address] [count [WHILE cond] [SILENT [count]]]
```

The TP command is alike the T command, but proceeds past repeated string instructions like the P command would.

5.33 TM command - Show or set Trace Mode

```
trace mode      TM [0|1]
```

5.34 TSR command - Enter TSR mode

```
enter TSR mode  TSR
```

5.35 U command - Disassemble

```
unassemble      U [range]
```

5.36 W command - Write Program

```
write program   W [address]
```

5.37 W command - Write Sectors

```
write sectors   W address drive sector count
```

5.38 X commands - Expanded Memory (EMS) commands

```
expanded mem    XA/XD/XM/XR/XS, X? for help
```

5.39 Y command - Run script file

```
run script      Y [partition/][scriptfile] [:label]
```

The Y command runs a script file. The script file is specified in two different ways, depending on whether the debugger is running as an 86-DOS application or as a boot-loaded kernel replacement.

- If running as an application, the script name is a regular pathname. It may be quoted with doublequotes if the pathname includes blanks. If the indicated drive supports long filenames (LFNs) then the debugger will first try to open the pathname as an LFN.
- Otherwise, the script name may start with a partition specification to use. (Refer to the ?BOOT help page in section 7.10 for partition specifications.) Then, the pathname relative to that partition's root directory follows. Long filenames are not supported. Note that it is not valid to run an empty script file when boot-loaded.

Further, a label may be specified to cause execution to start at that label instead of at the start of the file. This is equivalent to placing a '`GOTO :label`' command at the start of the script file. The colon to indicate a label is required.

If execution already is within a script file, then the Y command may be run with only a label (again with the colon required). In that case, the current script file is opened in a subsequent level (handle or boot-loaded script file context) and execution starts at that label.

Section 6: Variable Reference

6.1 Registers

All debuggee registers can be accessed numerically:

- `al`, `cl`, `dl`, `bl`, `ah`, `ch`, `dh`, `bh`
- `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`
- `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`
- `es`, `cs`, `ss`, `ds`, `fs`, `gs`
- `fl`, `efl`, `ip`, `eip`

Each 16-bit register can be used in a register pair, such as:

- `dxax`
- `bxcx` (used by `L` load program and `W` write program commands)
- `sidi`
- `csip`

6.2 Options

6.2.1 DCO - Debugger Common Options

6.2.2 DCS - Debugger Common Startup options

6.2.3 DIF - Debugger Internal Flags

6.2.4 DAO - Debugger Assembly Options

6.2.5 DAS - Debugger Assembly Startup options

6.2.6 DPI - Debugger Parent Interrupt 22h

6.2.7 DPR - Debugger PRocess

6.2.8 DPP - Debugger Parent Process

6.2.9 DPS - Debugger Process Selector

0 while in Real or Virtual 8086 Mode, debugger process selector otherwise. (The process selector addresses DebugX's PSP and DATA ENTRY section.)

6.3 Default step counts

PPC

Proceed command (section 5.25) default step count

TPC

Trace/Proceed command (section 5.32.1) default step count

TTC

Trace command (section 5.32) default step count

All of these are doublewords and default to 1. For the respective commands, these counts specify the number of steps to take if none is specified explicitly. This includes when a command is run by autorepeat, refer to section 5.1. If one of these is set to zero then it is an error to not specify a count explicitly for the corresponding command.

6.4 Limits

6.4.1 RELIMIT - RE buffer execution command limit

Doubleword. Default is 256. If this many commands are executed from the RE buffer, the execution is aborted and the command that called RE is continued.

6.4.2 RECOUNT - RE buffer execution command count

Doubleword. This is reset to zero when RE buffer execution starts. Each time a command is executed from the RE buffer, this variable is incremented. If it reaches the value of RELIMIT, RE buffer execution is aborted.

6.5 Return Codes

6.5.1 RC - Return Code

Word. This holds the most recent command's return code. If the most recent command succeeded, then this is zero.

6.5.2 ERC - Error Return Code

Word. This holds the most recent non-zero return code.

6.6 Addresses

6.6.1 A address (AAS:AAO)

AAS: word, AAO: doubleword. Default address for the assembler. Updated to point after each assembled instruction.

6.6.2 D address (ADS:ADO)

Default address for memory dumping. Updated to point after each dumped memory content.

6.6.3 Address behind R disassembly (ABS:ABO)

6.6.4 U address (AUS:AUO)

Default address for the disassembler.

6.6.5 E address (AES:AEO)

Default address for memory entry.

6.6.6 DZ address (AZS:AZO)

Default address for DZ command, ASCIZ strings. Terminated by zero byte.

6.6.7 D$ address (ACS:ACO)

Default address for D$ command, CP/M strings. Terminated by dollar sign '$'.

6.6.8 D# address (APS:APO)

Default address for D# command, Pascal strings. Prefixed by length count byte.

6.6.9 DW# address (AWS:AWO)

Default address for DW# command. Prefixed by length count word.

6.6.10 DX address (AXO)

Default address for DX command. (Only included in DebugX.)

6.7 I/O configuration

6.7.1 IOR - I/O Rows

Byte. Default 1. Sets the number of rows of the terminal used by DOS or BIOS output. Setting this to zero disables paging to the DOS or BIOS output. Setting this to 1 uses the automatic selection. That means the BIOS Data Area byte at address 484h, plus one, is used. If using that byte and it is zero, paging is disabled.

6.7.2 IOC - I/O Columns

Byte. Default 1. Sets the number of columns of the terminal used by BIOS input. Setting this to zero selects a default (80). Setting this to 1 uses the automatic selection. That means the BIOS Data Area word at address 44Ah is used. This is used by the line input handling if inputting from the BIOS terminal (int 16h, int 10h), or if inputting from a DOS terminal when DCO flag 800h is set.

6.8 Serial configuration

6.8.1 DSR - Debugger Serial Rows

Byte. Default 24. Sets the number of rows of the terminal connected via serial port. Setting this to zero disables paging to the serial port. Setting this to 1 uses the IOR variable handling.

6.8.2 DSC - Debugger Serial Columns

Byte. Default 80. Sets the number of columns of the terminal connected via serial port. Setting this to zero selects a default (80). Setting this to 1 uses the IOC variable handling. This is used by the line input handling.

6.8.3 DST - Debugger Serial Timeout

Byte. Default 15. This gives the number of seconds that the KEEP prompt upon serial connection waits. Setting this to zero waits at the prompt forever.

6.8.4 DSF - Debugger Serial FIFO size

Byte. Default 16. This gives the size of the 16550A's built-in TX FIFO to use. Set to 15 if using dosemu before revision gc7f5a828 2019-01-22, see https://github.com/stsp/dosemu2/issues/748.

6.8.5 DSPVI - Debugger Serial Port Variable Interrupt number

Byte. Default 0Bh, corresponding to COM2. Use 0Ch for COM1. This specifies the interrupt number to hook so as to be notified of serial events. The use of this variable occurs only when connecting to serial I/O. The value at that point in time is cached for as long as the serial connection is in use.

6.8.6 DSPVM - Debugger Serial Port Variable IRQ Mask

Word. Default 0000_1000b, corresponding to COM2. Use 0001_0000b for COM1. This specifies the IRQ mask of which IRQs to enable. The low 8 bits correspond to IRQ #0 to #7 and the high 8 bits correspond to IRQ #8 to #15. If any bit of the high 8 bits is set then generally the bit 0100b should be set too, to enable the chained PIC. This circumstance is not automatically detected. The use of this variable occurs only when connecting to serial I/O. The value at that point in time is cached for as long as the serial connection is in use.

6.8.7 DSPVP - Debugger Serial Port Variable base Port

Word. Default 02F8h, corresponding to COM2. Use 03F8h for COM1. This specifies the I/O port base to address the UART. The use of this variable occurs only when connecting to serial I/O. The value at that point in time is cached for as long as the serial connection is in use.

6.8.8 DSPVD - Debugger Serial Port Variable Divisor latch

Word. Default 12, corresponding to 9600 baud. This specifies the DL value to set during initialisation. The use of this variable occurs only when connecting to serial I/O.

6.8.9 DSPVS - Debugger Serial Port Variable Settings

Byte. Default 0000_0011b, corresponding to 8n1. (8n1 = 8 data bits, no parity, 1 stop bit.) This specifies the settings to set up in LCR. The high bit (80h) generally must be clear. The use of this variable occurs only when connecting to serial I/O.

6.8.10 DSPVF - Debugger Serial Port Variable FIFO select

Byte. Default 0. This specifies what to write to the FCR. The low 3 bits (07h) generally must

be clear. The use of this variable occurs only when connecting to serial I/O. The value at that point in time is cached for as long as the serial connection is in use.

6.9 _DEBUG1 variables

These variables are not supported by default. The build option _DEBUG1 must be enabled to include them. The Test Counter variables work similarly to permanent breakpoint counters:

- If the counter AND-masked with 7FFFh is zero, it is at a terminal state.
- If the counter is not yet at a terminal state, it is decremented.
- If the counter is decremented to zero, it triggers.
- If the counter is decremented to 8000h or already at 8000h, it triggers.

The default values for all counters and addresses is zero.

6.9.1 TRx - Test Readmem variables

If a fault is injected into readmem, it returns the value given in TRV.

TRC - Test Readmem Counter

Word. Each of the TRC0 to TRCF counters gives one counter for readmem fault injection testing.

TRA - Test Readmem Address

Doubleword. Each of the TRA0 to TRAF counters gives one linear address for readmem fault injection testing.

TRV - Test Readmem Value

Byte. Default 0. If a readmem fault is injected, this byte value is returned by the read instead of the actual memory content.

6.9.2 TWx - Test Writemem variables

If a fault is injected into writemem, it returns failure (CY).

TWC - Test Writemem Counter

Word. Each of the TWC0 to TWCF counters gives one counter for writemem fault injection testing.

TWA - Test Writemem Address

Doubleword. Each of the TWA0 to TWAF counters gives one linear address for writemem fault injection testing.

6.9.3 TLx - Test getLinear variables

If a fault is injected into getlinear, it returns failure (CY).

TLC - Test getLinear Counter

Word. Each of the TLC0 to TLCF counters gives one counter for getlinear fault injection testing.

TLA - Test getLinear Address

Doubleword. Each of the TLA0 to TLAF counters gives one linear address for getlinear fault injection testing.

6.9.4 TSx - Test getSegmented variables

If a fault is injected into getsegmented, it returns failure (CY).

TSC - Test getSegmented Counter

Word. Each of the TSC0 to TSCF counters gives one counter for getsegmented fault injection testing.

TSA - Test getSegmented Address

Doubleword. Each of the TSA0 to TSAF counters gives one linear address for getsegmented fault injection testing.

6.10 _DEBUG3 variables

These variables are not supported by default. The build option _DEBUG3 must be enabled to include them. These variables are used to test the read-only masking. Read-only masking makes it so that bits given in the mask are read-only. Bits that are clear in the mask are writable.

6.10.1 MT0 - Mask Test 0

Doubleword. Default 0. Mask AA55_AA55h.

6.10.2 MT1 - Mask Test 1

Doubleword. Default 0011_0022h. Mask 00FF_00FFh.

6.11 Y command variables

Y command variables can be used when the Y command (as application or bootloaded) has been used to open a script file. YSx (Y Script) variables are generic and refer to whatever Y file is opened. YBx (Y Bootloaded script) variables refer to opened Y files while bootloaded. YHx (Y Handle script) variables refer to opened Y files as application.

6.11.1 YSF - Y Script Flags

Word. Partially read-write, partially read-only.

Flag 4000h controls whether script file input is displayed or not. Prepending an AT sign (@) to a line that is read from a script file will hide the input of that line. Setting YSF flag 4000h will hide all input lines instead. The effect is similar to prepending @ to every line.

YSF variables are only available while executing script files.

6.12 V variables - Variables with user-defined purpose

Doubleword. Default zero. V0 to VF or V00 to VFF each specify one variable. It is valid to refer to any V variable using an index expression. Index expression means that the variable name (V) is immediately followed by an opening parenthesis, followed by a numeric expression which evaluates to a number below 100h.

6.13 PSP variables

6.13.1 PSP - Process Segment Prefix

6.13.2 PPR - Process PaRent

6.13.3 PPI - Process Parent Interrupt 22h

6.14 SR variables - Search Results

6.14.1 SRC - Search Result Count

Doubleword. Read only. Amount of matches found by last S command.

6.14.2 SRS - Search Result Segment

Word. Read only. SRS0 to SRSF each specify one variable. Search result segments of last S command's matches.

6.14.3 SRO - Search Result Segment

Word or doubleword (DebugX). Read only. SRO0 to SROF each specify one variable. Search result offsets of last S command's matches. It is valid to refer to any SRO variable using an index expression. Index expression means that the variable name (SRO) is immediately followed by an opening parenthesis, followed by a numeric expression which evaluates to a number below 10h.

Section 7: Online help pages

7.1 ? - Main online help

```
lDebug (2020-03-25) help screen
assemble        A [address]
set breakpoint  BP index|AT|NEW address
                  [[NUMBER=]number] [WHEN=cond] [ID=id]
 set ID         BI index|AT address [ID=]id
 set condition  BW index|AT address [WHEN=]cond
 set offset     BO index|AT address [OFFSET=]number
 set number     BN index|AT address|ALL number
 clear          BC index|AT address|ALL
 disable        BD index|AT address|ALL
 enable         BE index|AT address|ALL
 toggle         BT index|AT address|ALL
 list           BL [index|AT address|ALL]
compare         C range address
dump            D [range]
dump bytes      DB [range]
dump words      DW [range]
dump dwords     DD [range]
dump interrupts DI interrupt [count]
dump MCB chain  DM [segment]
display strings DZ/D$/D[W]# [address]
enter           E address [list]
fill            F range [RANGE range|list]
go              G [=address] [breakpts]
goto            GOTO :label
hex add/sub     H value1 [value2 [...]]
input           I[W|D] port
if numeric      IF [NOT] (cond) THEN cmd
if script file  IF [NOT] EXISTS Y file [:label] THEN cmd
load program    L [address]
load sectors    L address drive sector count
move            M range address
80x86/x87 mode  M [0..6|C|NC|C2|?]
set name        N [[drive:][path]progname.ext [parameters]]
output          O[W|D] port value
proceed         P [=address] [count [WHILE cond] [SILENT [count]]]
quit            Q
register        R [register [value]]
R extended      RE
```

```
RE commands     RE.LIST|APPEND|REPLACE [commands]
MMX register    RM
FPU register    RN
toggle 386 regs RX
search          S range [REVERSE] [RANGE range|list]
trace           T [=address] [count [WHILE cond] [SILENT [count]]]
trace (exc str) TP [=address] [count [WHILE cond] [SILENT [count]]]
trace mode      TM [0|1]
enter TSR mode  TSR
unassemble      U [range]
write program   W [address]
write sectors   W address drive sector count
expanded mem    XA/XD/XM/XR/XS, X? for help
run script      Y [partition/][scriptfile] [:label]

Additional help topics:
 Registers      ?R
 Flags          ?F
 Conditionals   ?C
 Expressions    ?E
 Variables      ?V
 R Extended     ?RE
 Run keywords   ?RUN
 Options        ?O
 Boot loading   ?BOOT
 lDebug build   ?BUILD
 lDebug build   ?B
 lDebug sources ?SOURCE
 lDebug license ?L
```

7.2 ?R - Registers

```
Available 16-bit registers:             Available 32-bit registers: (386+
AX      Accumulator                     EAX
BX      Base register                   EBX
CX      Counter                         ECX
DX      Data register                   EDX
SP      Stack pointer                   ESP
BP      Base pointer                    EBP
SI      Source index                    ESI
DI      Destination index               EDI
DS      Data segment
ES      Extra segment
SS      Stack segment
CS      Code segment
FS      Extra segment 2 (386+)
GS      Extra segment 3 (386+)
IP      Instruction pointer             EIP
FL      Flags                           EFL
```

Available 64-bit Matrix Math Extension (MMX) registers: (if supported)

```
MMx     MM(x)   MMX register x, where x is 0 to 7
```

Enter ?F to display the recognized flags.

7.3 ?F - Flags

Recognized flags:

```
Value   Name                    Set                   Clear
0800  OF  Overflow Flag       OV  Overflow           NV  No overflow
0400  DF  Direction Flag      DN  Down               UP  Up
0200  IF  Interrupt Flag      EI  Enable interrupts  DI  Disable inter
0080  SF  Sign Flag           NG  Negative           PL  Plus
0040  ZF  Zero Flag           ZR  Zero               NZ  Not zero
0010  AF  Auxiliary Flag      AC  Auxiliary carry    NA  No auxiliary
0004  PF  Parity Flag         PE  Parity even        PO  Parity odd
0001  CF  Carry Flag          CY  Carry              NC  No carry
```

The short names of the flag states are displayed when dumping registers and can be entered to modify the symbolic F register with R. The short names of the flags can be modified by R.

7.4 ?C - Conditionals

In the register dump displayed by the R, T, P and G commands, conditional jumps are displayed with a notice that shows whether the instruction will cause a jump depending on its condition and the current register and flag contents. This notice shows either "jumping" or "not jumping" as appropriate.

The conditional jumps use these conditions: (second column negates)

```
 jo             jno             OF
 jc jb jnae     jnc jnb jae     CF
 jz je          jnz jne         ZF
 jbe jna        jnbe ja         ZF||CF
 js             jns             SF
 jp jpe         jnp jpo         PF
 jl jnge        jnl jge         OF^^SF
 jle jng        jnle jg         OF^^SF || ZF
 j(e)cxz                        (e)cx==0
 loop                           (e)cx!=1
 loopz loope                    (e)cx!=1 && ZF
 loopnz loopne                  (e)cx!=1 && !ZF
```

Enter ?F to display a description of the flag names.

7.5 ?E - Expressions

Recognized operators in expressions:

```
|       bitwise OR              ||      boolean OR
^       bitwise XOR             ^^      boolean XOR
&       bitwise AND             &&      boolean AND
>>      bit-shift right         >       test if above
>>>     signed bit-shift right  <       test if below
```

```
<<      bit-shift left          >=      test if above-or-equal
><      bit-mirror              <=      test if below-or-equal
+       addition                ==      test if equal
-       subtraction             !=      test if not equal
*       multiplication          =>      same as >=
/       division                =<      same as <=
%       modulo (A-(A/B*B))      <>      same as !=
**      power
```

Implicit operater precedence is handled in the listed order, with increasing precedence: (Brackets specify explicit precedence of an expression.)

```
boolean operators OR, XOR, AND (each has a different precedence)
comparison operators
bitwise operators OR, XOR, AND (each has a different precedence)
shift and bit-mirror operators
addition and subtraction operators
multiplication, division and modulo operators
power operator
```

Recognized unary operators: (modifying the next number)

```
+       positive (does nothing)
-       negative
~       bitwise NOT
!       boolean NOT
?       absolute value
!!      convert to boolean
```

Note that the power operator does not affect unary operator handling. For instance, "- 2 ** 2" is parsed as "(-2) ** 2" and evaluates to 4.

Although a negative unary and signed bit-shift right operator are provided the expression evaluator is intrinsically unsigned. Particularly the division, multiplication, modulo and all comparison operators operate unsigned. Due to this, the expression "-1 < 0" evaluates to zero.

Recognized terms in an expression:

```
32-bit immediates
8-bit registers
16-bit registers including segment registers (except FS, GS)
32-bit compound registers made of two 16-bit registers (eg DXAX)
32-bit registers and FS, GS only if running on a 386+
32-bit variables V00..VFF
32-bit special variables DCO, DCS, DAO, DAS, DIF, DPI, PPI
16-bit special variables DPR, DPP, PSP, PPR
 (fuller variable reference in the manual)
byte/word/3byte/dword memory content (eg byte [seg:ofs], where both the
 optional segment as well as the offset are expressions too)
```

The expression evaluator case-insensitively checks for names of variables and registers as well as size specifiers.

Enter ?R to display the recognized register names. Enter ?V to display the recognized variables.

7.6 ?V - Variables

Available lDebug variables:

- V0..VF User-specified usage
- DCO Debugger Common Options
- DAO Debugger Assembler/disassembler Options

The following variables cannot be written:

- PSP Debuggee Process
- PPR Debuggee's Parent Process
- PPI Debuggee's Parent Process Interrupt 22h
- DIF Debugger Internal Flags
- DCS Debugger Common Startup options
- DAS Debugger Assembler/disassembler Startup options
- DPR Debugger Process
- DPP Debugger's Parent Process (zero in TSR mode)
- DPI Debugger's Parent process Interrupt 22h (zero in TSR mode)

Enter ?O to display the options and internal flags.

7.7 ?RE - R Extended

The RUN commands (T, TP, P, G) and the RE command use the RE command buffer to run commands. Most commands are allowed to be run from the RE buffer. Disallowed commands include program-loading L, A, E that switches the line input mode, TSR, Q, Y, RE, and further RUN commands. When the RE buffer is used as input during T, TP, or P with either of the WHILE or SILENT keywords, commands that use the auxbuff are also disallowed and will emit an error noting the conflict.

RE.LIST shows the current RE buffer contents in a format usable by the other RE commands. RE.APPEND appends the following commands to the buffer, if they fit. RE.REPLACE appends to the start of the buffer. When specifying commands, an unescaped semicolon is parsed as a linebreak to break apart individual commands. Backslashes can be used to escape semicolons and backslashes themselves.

Prefixing a line with an @ (AT sign) causes the command not to be shown to the standard output of the debugger when run. Otherwise, the command will be shown with a percent sign % or ~% prompt.

The default RE buffer content is @R. This content is also detected and handled specifically; if found as the only command the handler directly calls the register dump implementation without setting up and tearing down the special execution environment used to run arbitrary commands from the RE buffer.

7.8 ?RUN - Run keywords

T (trace), TP (trace except proceed past string operations), and P (proceed) can be followed by a number of repetitions and then the keyword WHILE, which must be followed by a conditional expression.

The selected run command is repeated as many times as specified by the number, or until the WHILE condition evaluates no longer to true.

After the number of repetitions or (if present) after the WHILE condition the keyword SILENT may follow. If that is the case, all register dumps done during the run are buffered by the debugger and the run remains silent. After the run, the last dumps are replayed from the buffer and displayed. At most as many dumps as fit into the buffer are displayed. (The buffer is currently up to 8 KiB sized.)

If a number follows behind the SILENT keyword, only at most that many dumps are displayed from the buffer. The dumps that are displayed are always those last written into the buffer, thus last occurred.

7.9 ?O - Options

Available options: (read/write DCO, read DCS)

- 0001 RX: 32-bit register display
- 0002 TM: trace into interrupts
- 0004 allow dumping of CP-dependant characters
- 0008 always assume InDOS flag non-zero, to debug DOS or TSRs
- 0010 disallow paged output to StdOut
- 0020 allow paged output to non-StdOut
- 0040 display raw hexadecimal content of FPU registers
- 0100 when prompting during paging, do not use DOS for input
- 0200 do not execute HLT instruction to idle
- 0400 do not idle, the keyboard BIOS idles itself
- 0800 use rawinput for int 21h interactive input
- 1000 in disp_*_size use SI units (kB = 1000, etc). overrides 2000!
- 2000 in disp_*_size use JEDEC units (KB = 1024)
- 4000 enable serial I/O (port 02F8h interrupt 0Bh)
- 8000 disable serial I/O when breaking after 5 seconds Ctrl pressed
- 00010000 gg: do not skip a breakpoint (bb or gg)
- 00020000 gg: do not auto-repeat

- 00040000 T/TP/P: do not skip a (bb) breakpoint
- 00080000 gg: do not auto-repeat after bb hit
- 00100000 T/TP/P: do not auto-repeat after bb hit
- 00200000 gg: do not auto-repeat after unexpectedinterrupt
- 00400000 T/TP/P: do not auto-repeat after unexpectedinterrupt
- 00800000 S: do not dump data after matches
- 10000000 R: do not repeat disassembly
- 20000000 R: do not show memory reference in disassembly
- 40000000 quiet command line buffer input
- 80000000 quiet command line buffer output

More options: (read/write DCO2, read DCS2)

- 0001 DB: show header
- 0002 DB: show trailer
- 0010 DW: show header
- 0020 DW: show trailer
- 0100 DD: show header
- 0200 DD: show trailer
- 0800 use rawinput for int 21h interactive input in DPMI
- 1000 H: stay compatible to MS-DOS Debug
- 2000 idle and check for Ctrl-C in getc
- 4000 idle and check for Ctrl-C in getc in DPMI
- 8000 T/TP/P/G: cancel run after RE command buffer execution

More options: (read/write DCO3, read DCS3)

- 0001 T: do not page output
- 0002 TP: do not page output
- 0004 P: do not page output
- 0008 G: do not page output
- 0100 T/TP/P: modify paging for silent dump
- 0200 T/TP/P: if 0100 set: turn paging on, else off

Internal flags: (read DIF)

- 000001 Int25/Int26 packet method available
- 000002 Int21.7305 packet method available
- 000004 VDD registered and usable
- 000008 internal flag for paged output
- 000010 DEBUG's input isn't StdIn
- 000020 DEBUG's input is a file
- 000040 DEBUG's output isn't StdOut
- 000080 DEBUG's output is a file
- 001000 state of debuggee's A20
- 002000 state of debugger's A20 (not implemented: same as previous)
- 004000 debugger booted independent of a DOS
- 008000 CPU is at least a 386 (32-bit CPU)
- 010000 internal flag for tab output processing
- 020000 running inside NTVDM
- 100000 internal flag for paged output
- 400000 in TSR mode (detached debugger process)
- 01000000 running inside dosemu
- 04000000 T/TP/P: while condition specified
- 08000000 TP: P specified (proceed past string ops)
- 10000000 T/TP/P: silent mode (SILENT specified)
- 20000000 T/TP/P: silent mode is active, writing to silent buffer

Available assembler/disassembler options: (read/write DAO, read DAS)

- 01 Disassembler: lowercase output
- 02 Disassembler: output blank behind comma
- 04 Disassembler: output addresses in NASM syntax
- 08 Disassembler: lowercase referenced memory location segreg
- 10 Disassembler: always show SHORT keyword
- 20 Disassembler: always show NEAR keyword
- 40 Disassembler: always show FAR keyword

7.10 ?BOOT - Boot loading

Boot loading commands:

- BOOT LIST HDA [note: writes to memory @ 600h and 7C00h]
- BOOT READ|WRITE [partition] segment [[HIDDEN=sector] sector] [count]
- BOOT QUIT [exits dosemu or shuts down using APM]
- BOOT [PROTOCOL=SECTOR] partition
- BOOT PROTOCOL=proto [opt] [partition] [filename1] [filename2] [cmdline]
- the following partitions may be specified:
    - HDAnum first hard disk, num = partition (1-4 primary, 5+ logical)
    - HDBnum second hard disk (etc), num = partition
    - HDA first hard disk (only valid for READ|WRITE|PROTOCOL=SECTOR)
    - FDA first floppy disk
    - FDB second floppy disk (etc)
    - LDP partition the debugger loaded from
    - YDP partition the most recent Y command loaded from
    - SDP last used partition (default if no partition specified)
    - filename2 may be double-slash // for none
    - cmdline is only valid for lDOS, RxDOS.2, RxDOS.3 protocols
    - files' directory entries are loaded to 500h and 520h

Available protocols: (default filenames, load segment, then entrypoint)

- LDOS LDOS.COM or L[D]DEBUG.COM at 200h, 0:400h
- FREEDOS KERNEL.SYS or METAKERN.SYS at 60h, 0:0
- DOSC IPL.SYS at 2000h, 0:0
- EDRDOS DRBIO.SYS at 70h, 0:0
- MSDOS6 IO.SYS + MSDOS.SYS at 70h, 0:0
- MSDOS7 IO.SYS at 70h, 0:200h
- IBMDOS IBMBIO.COM + IBMDOS.COM at 70h, 0:0
- NTLDR NTLDR at 2000h, 0:0
- BOOTMGR BOOTMGR at 2000h, 0:0

- RXDOS.0 RXDOSBIO.SYS + RXDOS.SYS at 70h, 0:0
- RXDOS.1 RXBIO.SYS + RXDOS.SYS at 70h, 0:0
- RXDOS.2 RXDOS.COM at 70h, 0:400h
- RXDOS.3 RXDOS.COM at 200h, 0:400h
- CHAIN BOOTSECT.DOS at 7C0h, -7C0h:7C00h
- SECTOR (default) load partition boot sector or MBR
- SECTORALT as SECTOR, but entry at 07C0h:0

Available options:

- MINPARA=num load at least that many paragraphs
- MAXPARA=num load at most that many paragraphs (0 = as many as fit)
- SEGMENT=num change segment at that the kernel loads
- ENTRY=[num:]num change entrypoint (CS (relative) : IP)
- BPB=[num:]num change BPB load address (segment -1 = auto-BPB)
- CHECKOFFSET=num set address of word to check, must be even
- CHECKVALUE=num set value of word to check (0 = no check)

Boolean options: [opt=bool]

- SET_DL_UNIT set dl to load unit
- SET_BL_UNIT set bl to load unit
- SET_SIDI_CLUSTER set si:di to first cluster
- SET_DSSI_DPT set ds:si to DPT address
- PUSH_DPT push DPT address and DPT entry address
- DATASTART_HIDDEN add hidden sectors to datastart var
- SET_AXBX_DATASTART set ax:bx to datastart var
- SET_DSBP_BPB set ds:bp to BPB address
- LBA_SET_TYPE set LBA partition type in BPB
- MESSAGE_TABLE provide message table pointed to at 1EEh
- SET_AXBX_ROOT_HIDDEN set ax:bx to root start with hidden sectors
- NO_BPB do not load BPB
- SET_DSSI_PARTINFO load part table to 600h, point ds:si + ds:bp to it

7.11 ?BUILD - lDebug build (only revisions)

```
lDebug (YYYY-MM-DD)
Source Control Revision ID: hg xxxxxxxxxxxx
Uses yyyyyyyy: Revision ID hg zzzzzzzzzzzz
[etc]
```

7.12 ?B - lDebug build (with options)

```
lDebug (YYYY-MM-DD)
Source Control Revision ID: hg xxxxxxxxxxxx
Uses yyyyyyyy: Revision ID hg zzzzzzzzzzzz
[etc]

DI command
DM command
D string commands
S match dumps line of following data
RN command
Access SDA current PSP field
Load NTVDM VDD for sector access
X commands for EMS access
RM command and reading MMX registers as variables
Expression evaluator
 Indirection in expressions
Variables with user-defined purpose
Debugger option and status variables
PSP variables
Conditional jump notice in register dump
TSR mode (Process detachment)
Boot loader
Permanent breakpoints
Intercepted interrupts: 00, 01, 03, 06, 18, 19
Extended built-in help pages
```

7.13 ?X - EMS commands

```
Expanded memory (EMS) commands:
  Allocate      XA count
  Deallocate    XD handle
  Map memory    XM logical-page physical-page handle
  Reallocate    XR handle count
  Show status   XS
```

7.14 ?SOURCE - lDebug source reference

The original lDebug releases can be obtained from the repo located at https://hg.ulukai.org/ecm/ldebug (E. C. Masloch's repo)

The most recent manual is hosted at https://ulukai.org/ecm/doc/ in the files ldebug.htm, ldebug.txt, and ldebug.pdf

7.15 ?L - lDebug license

lDebug - libre 86-DOS debugger

- Copyright (C) 1995-2003 Paul Vojta
- Copyright (C) 2008-2021 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.

All contributions by Paul Vojta or C. Masloch to the debugger are available under a choice of three different licenses. These are the Fair License, the Simplified 2-Clause BSD License, or the MIT License.

This is the license and copyright information that applies to lDebug; but note that there have been substantial contributions to the code base that are not copyrighted (public domain).

Section 8: Additional usage conditions

The program executables can be compressed with a choice of different compressors. The files then contain a decompression stub. Some of these stubs have their own usage conditions. The following stub usage conditions apply, if one of these stubs is used.

8.1 BriefLZ depacker usage conditions

BriefLZ - small fast Lempel-Ziv

8086 Assembly lDOS iniload payload BriefLZ depacker

Based on: BriefLZ C safe depacker

Copyright (c) 2002-2016 Joergen Ibsen

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

8.2 LZ4 depacker usage conditions

8086 Assembly lDOS iniload payload LZ4 depacker

by C. Masloch, 2018

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.

8.3 Snappy depacker usage conditions

8086 Assembly lDOS iniload payload Snappy depacker

by C. Masloch, 2018

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.

8.4 Exomizer depacker usage conditions

8086 Assembly lDOS iniload payload exomizer raw depacker

by C. Masloch, 2020

Copyright (c) 2005-2017 Magnus Lind.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented * you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any distribution.

4. The names of this software and/or it's copyright holders may not be used to endorse or promote products derived from this software without specific prior written permission.

8.5 X compressor depacker usage conditions

MIT License

Copyright (c) 2020 David Barina

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

8.6 Heatshrink depacker usage conditions

8086 Assembly lDOS iniload payload heatshrink depacker

by C. Masloch, 2020

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.

8.7 Lzd usage conditions

Lzd - Educational decompressor for the lzip format

Copyright (C) 2013-2019 Antonio Diaz Diaz.

This program is free software. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

8.8 LZO depacker usage conditions

8086 Assembly lDOS iniload payload LZO depacker

by C. Masloch, 2020

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.

Source Control Revision ID

hg 5f34bad78830, from commit on at 2021-02-15 12:26:03 +0100

If this is in ecm's repository, you can find it at https://hg.ulukai.org/ecm/ldebug/rev/5f34bad78830