lDebug manual

This document has been compiled on 2022-09-24.

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 option being developed.

Section 2: News

2.1 Release 5 (future)

- Add BS command for swapping permanent breakpoint indices
- Document doubled delimiter quote mark for lists and string literals
- Add string literal escaping of delimiter quote mark by doubling the delimiter quote mark
- Add `/2` switch to use alternative video adapter for debugger output if available (pick from FreeDOS Debug)
- Add `?OPTIONS` help page and specific pages for DCO1, DCO2, DCO3, DCO4, DCO6, DIF, and DAO
- Set new `INICOMP_WINNER` build variable so as to use lzsa2 compression for current releases
- Add `_DEBUG_COND` build option to allow toggling debug mode on and off at run time
- Add `INT8CTRL` variable which contains number of ticks to wait for Control pressed entrypoint; set to zero to disable
- Fix: Control-C also aborts RC command buffer execution
- Fix: Default operand for `AAM` and `AAD` instructions is omitted in disassembler
- Enhancement: If at the end of a stdin-redirected file the debugger cannot quit it will now enable InDOS mode and allow the user to control the debugger afterwards
- Fix: Do not crash or loop infinitely upon encountering the end of a stdin-redirected file
- Extract more source files from debug.asm
- Allow appending `00` to a 16-bit register name to get a 32-bit value with the register value in the high word
- Do not cause error from empty `/C=''` switch
- Use ampersand prompt to display commands run from RC buffer
- When loading a .BIN file set the process's command line buffer the same way as if loading a .COM file
- Add heading hash links to every heading in the ldebug.htm manual (requires patched Halibut)
- Add LFSR and LFSRTAP variables

- Run unix2dos on ldebug.txt manual
- Add QD (quit from device initialisation) and QC (quit from device in container MCB) commands
- Add RVD command to display device header address and allocation size, as well as DEVICEHEADER and DEVICESIZE variables to read same
- Bugfix, on pass or non-pass permanent breakpoint hit while running with T/TP/P command do not check WHILE condition
- Add `PARAS` keyword to range length parsing, to multiply a count by 16 (size of a paragraph)
- Bugfix, should allow to run if int 2Fh is invalid
- Add device-driver mode to allow loading the debugger in CONFIG.SYS
- Fix, do not crash if no UMCB but int 21.5803 works
- Add V commands and /V command-line switch (video screen swapping)
- Add `RIxxP` variables to read IVT entries in a way suitable to be used as `POINTER` type expressions
- Work around FreeDOS kernel bug prior to 2022 May so as to fail on loading an empty executable
- Fix, also use SDA manipulation to change current PSP when lDebugX is in Protected Mode
- Add TERMCODE variable to read int 21.4D return after debuggee process terminated
- Add QB command (run breakpoint late in debugger quit)
- Add RVP command to display debugger mode and current debuggee and debugger process addresses
- Add (D)PSP|PARENT|PRA|PSPSEL variables
- Do not try to proceed past a call near immediate if the called functions consists of a `retf` instruction. (This supports a method for relocation, used for example by the debugger itself.)
- Add command-line switch /B to run a breakpoint early
- Add RC commands to view, change, and run RC buffer commands, re-using the command line buffer
- Add `MACHX86` and `MACHX87` variables to read machine type
- Allow M machine type command to parse an expression for the machine level number to set
- Add QA command (try to terminate attached process)
- Fix int 19h and debuggee termination handling. Int 19h in a DOS application mode now sets up registers to terminate the current process when running the debuggee again.

- Add an lDebugX option DCO3 20_0000 to break on entering PM
- Add an lDebugX option DCO3 10_0000 to use a 32-bit stack segment for the debugger itself (can help compatibility)
- Fix so that semicolon is allowed as End Of Line in getrange
- Fix `R size [mem] := val` causing a fault in the debugger if value ends in FFFFh
- Implement `POINTER` types for handling a 32-bit expression as a 16:16 far pointer
- Implement basic handling of expression types (signed/unsigned)
- Revision IDs in `?BUILD` command list the amount of ancestors to help to compare revisions
- Fix a segment addressing bug when switching modes (eg have a breakpoint in a DPMI allocation while the client is running in 86 Mode)
- Fix some cases of detecting 32-bit offsets incorrectly

2.2 Release 4 (2022-03-08)

- Recognise LF as linebreak in serial input
- E interactive mode fixes:
  - Support LF to exit interactive mode (that is, accept Linux style linebreaks)
  - Support DEL sent by serial terminal
  - In lDebugX correctly handle 32-bit offsets
  - Also write new value when minus is entered
  - Honour blank for continue to next byte, CR or dot for exit interactive mode
  - Always correctly read value even if blank is entered afterwards
  - Improve E interactive mode compatibility across different input sources (like stdin file, script file, serial terminal)
  - Display linebreak upon new address displayed
- Fix: Register variable '`CH`' would be misparsed as '`CHAR`' type instead of the expected variable
- Allow DI command to receive an IN value list similar to the y in a VALUE x IN y construct
- Fix: Allow to set a breakpoint on an interrupt 21h handler and do not crash or corrupt state if the debuggee then terminates. (That is, do not call service 4Dh before restoring breakpoints.)
- Fix: Too long N command could crash the debugger
- Fix: DDebug TSR quit would not work correctly due to overflowing a rel8 `jmp`
- Add R, M, and L key letters to DI command (always 86 Mode, show MCB names, follow

AMIS interrupt lists)

- Fix: `R WORD [memory]` prompt would not consider the size keyword as part of the input line prompt
- Add AMIS private function 30h - Update IISP Header
- In DI command in 86 Mode follow IISP headers
- Add `QQCODE` variable
- Add `BOOT[L|Y|S][UNIT|PART]` variables, `BOOTUNITFL(x)` variables
- Add bzpack compression method
- Drop DPS variable when building without DPMI support
- Fix PSP variables in Protected Mode: PSP is always a 86 Mode segment, PSPS is a segment or selector, and PPR and PPI work
- Add `HHRESULT` variable

2.3 Release 3 (2021-08-15)

- Add workaround with extra int 23h and int 22h handlers and raw mode-switching to use interrupt 21h service 0Ah in PM. DCO2 flag 800h clear by default.
- Add TRYAMISNUM variable to try a specific AMIS multiplex number first
- Add DCO4 flag 2 to allow disabling lDebugX's int 2Fh hook
- Build option `_MEMREF_AMOUNT` enabled by default
- mktables switches `direction` and `stackhinting` enabled by default
- Fix DOS application script file reading to honour InDOS status
- Fix H BASE= command with GROUP= sometimes displaying trailing garbage
- Fix DDebugX hooking random PM interrupts
- Fix trailing blanks in DI command
- Added a number of automated acceptance tests
- Add variable `AMISNUM` to read the multiplex number
- Fix an old bug in the assembler that happened to make instructions like '`mov ax, 0`' fail to assemble now
- Made interrupt 8 hook optional, default-off
- Added optional, default-off interrupt 2Dh hook
- Properly unhook interrupts utilising IISP header chains, if the debugger's interrupt handlers are reachable. Added DCO4 flags (upper 16 bits) to force unhooking if a handler is unreachable. If a handler is both unreachable and not forcibly unhooked then it stays hooked. The Q command fails in that case.

- Fix to allow '$' prefix to segments in DebugX while in Real/Virtual 86 Mode
- Debugger's 86 Mode entrypoints now use the IBM Interrupt Sharing Protocol header. (However, it is still assumed that the debugger *owns* the interrupt entrypoints.)
- Add `WIDTH=` keyword handling to `H BASE=`
- Introduce variables IOL and IOF to control how many levels of execution are cancelled by Control-C
- Scripts with CR LF linebreaks at the end or after calling another script no longer cause superfluous empty lines to be processed
- Control-C aborts script file reading that is in progress
- Bugfix, when calling three nested levels of Y script files while bootloaded then the outermost script's already buffered content would not rewind properly
- Fix so that Control-C from ROM-BIOS keypress buffer is consumed properly while reading script file, instead of looping forever
- Check for Control-C in ROM-BIOS's circular keypress buffer, add variables IOS and IOE
- Extend Control-C handling so RE buffer execution is aborted by it
- Add a simple `BOOT DIR` command (SFN name only, attributes, size (using FAT+), datetime)
- Add string literals `#"..."` to expression evaluator
- Add `H BASE=` command
- Add `merge` and `debug` switches to mktables. Both are default off for now. Merging means redundant operand list tails are merged.
- Bugfix, accessing the variable SRC caused an infinite loop
- LZMA-lzip depacker fixed to not use `cs xlatb`, as the segment override prefix may be ignored on CPUs below 386
- Added conditional `?? ::` construct operator
- Merged branch `uumemref` and made memrefs available in default branch. The build option `_MEMREF_AMOUNT` must be enabled to use them.
- Memory access direction and stack hinting in the assembler and disassembler tables. Switches named `direction` and `stackhinting` to mktables program. (Default off for now.)
- LINEAR term allowed in expressions
- VALUE IN construct allowed in expressions
- Commas are only allowed between expressions, no longer within expressions
- If DCO2 flag 8000h is set during RE buffer execution and SILENT 1 was used do actually only display last RE output

2.4 Release 2 (2021-05-05)

- Documented SLEEP command
- Line editing history for raw terminal/serial input (in a fixed segment of size 8 KiB currently)
- Fix missing register dump after T/TP/P which ends up matching a non-pass non-hit breakpoint
- Fix: Entering a literal as 3#102002022201221111211 or #4294967296 would overflow silently to zero instead of causing an error
- Reset high words of EIP and ESP when trying to terminate client process
- Add change highlighting to R register dump
- Assembler internals: Allow ASM_ESCAPE usage when needed
- If BL command is given an unused index do not display incorrect WHEN
- Reset segment registers when trying to terminate client process
- Handle unusual SIB bytes correctly in P command's disassembly
- Bugfix, Y script file called by another Y script file would turn quiet
- Bugfix, if permanent breakpoint WHEN condition was in use then the wrong index and ID would be displayed in the pass/hit message
- Acknowledge IRQ to secondary PIC too if applicable (if using a high IRQ for the serial I/O interrupt)
- Bugfix, in BOOT commands do not prepend a word to the auxbuff anymore
- Only create manual in HTML, text, and PDF formats
- Add files doc/fdbuild.txt and doc/LDEBUG.LSM for FreeDOS packages
- BOOT: work around qemu bug with '`LOOPNZ`'
- BOOT: retry CHS reads up to 16 times
- Add instsect and lDebug command help to manual
- Expression evaluator allows '`OR=`' as synonym for '`|=`' (especially useful if shell does not allow specifying pipe symbol for /C)
- Assembler: Allow specifying '`LOOPxx destination, (E)CX`' as in NASM instruction reference to specify address size
- For assembler allow specifying '`INT BYTE 3`' to get CDh encoding and display it this way in disassembler
- Only adjust offset saved in PSP's SPSAV variable if it points to our stack
- In assembler do not allow sizeless memory operand when immediate matches IMMS8 (eg '`add [100], 12`')

2.5 Release 1 (2021-02-15) and earlier

- ‘`G REMEMBER`’ command to work with the saved temporary breakpoint list
- WHEN conditions for permanent breakpoints
- RIxxO/S/L variables (read-only view of IVT entry)
- 3BYTE type for ‘`R var`’ and indirection in expression evaluator
- In disassembler handle unusual SIB byte contents correctly
- IDs for listing permanent breakpoints
- In disassembler correctly dump far memory operands, double memory operands (BOUND), and do a32 addressing
- Add ‘`S range REVERSE`’ command
- Fix corner case of S command: The commands ‘`f 100 l 10 0`’\‘`s 100 l 10 0`’ should result in 16 matches
- SROx and SRC search result variables
- SLEEP command
- H command displays decimal numeric value (when given a single expression)
- In disassembler display WORD keyword when o16 in 32-bit CS
- Bugfix, in XR do not skip first digit of allocation size
- G and T/TP/P breakpoints work reliably in DebugX when the client enters, leaves, or switches from/to Protected Mode
- F and S command allow accepting ‘`RANGE`’ specifications for source data
- Add TTC/TPC/PPC default step counts for T/TP/P commands
- DW/DD commands to dump memory in words or doublewords
- Manual added (this document)
- RE buffer execution to run almost arbitrary commands when T/TP/P/G intend to dump register contents
- Conditional control flow with IF and GOTO in a script file
- /C command line option to pass commands to the debugger on startup
- In assembler allow specifying SHORT/NEAR/FAR for jumps and calls
- Script file reading
- Pass point functionality (inspired by DR-DOS's SID) using counters
- G LIST command to list the saved temporary breakpoint list
- Auto-repetition for G command, G AGAIN command

- DebugX's DPMI entrypoint hooking automatically checked instead of always avoiding it on MSW and dosemu
- Serial port I/O, with defaults (for COM2) that can be reconfigured using debugger variables
- Permanent breakpoints
- Buffered tracing using '`P/TP/T ... SILENT`' which writes to an internal buffer during the run then replays the last entries from it upon finishing the run
- TP command which is like T except it handles repeated string operations like P
- DM command lists MCB sizes in decimal Bytes/KiB
- Conditional tracing using '`P/TP/T ... WHILE`' conditions
- L and W commands allow drive letters instead of numbers
- Bootloaded mode and its BOOT commands
- NASM style address disassembly, blanks after commas, keywords uncapitalised
- TSR mode and command to enter it
- R command allows treating flags (CF, ZF, etc), debugger variables, registers, and memory variables (byte, word, 3byte, dword) as variables
- Conditional "jumping" and "not jumping" notices in register dump's single-line disassembly
- Options DCO1, DCO2, DCO3, DAO to modify some behaviour
- Extended online help pages
- _DEBUG option which swaps the exception handlers and thus allows debugging most of the debugger itself (_DEBUG builds are not included in the package and have to be created by building them specifically)
- Arbitrary unsigned 32-bit expression evaluator
- Paging for long command output
- Usage conditions changed to Fair License (having asked Paul Vojta and received his confirmation), prior conditions also allowed as alternatives

Section 3: Building the debugger

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

3.1 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
- wc - to count amount of ancestors
- python - to run hg and to run the test suite
- 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 and a bug with %00

        - (As of 2022-08-23) Current git head with a patch for the %strcat bug and with a patch for the %00 bug works (last tested 2022-08)
- 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 or the test suite, 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 and create boot image for qemu to use for the test suite)
    - 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
        - lzsa
        - apultra
        - bzpack
    - crc16-t/iniload (if to add checksumming)
    - symsnip (only if symbolic option is enabled)

3.2 How to build

1. Clone the mercurial repo from https://hg.pushbx.org/ecm/ldebug or in an existing repo use '`hg pull`' to update the repo

2. Update the repo with '`hg up`' or '`hg up default`' or any other available commit you want to build

3. Clone the other needed repos from https://hg.pushbx.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 either as application or as device driver. (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.

3.2.1 How to build the instsect application

1. Clone the mercurial repo from https://hg.pushbx.org/ecm/ldebug or in an existing repo use '`hg pull`' to update the repo
2. Update the repo with '`hg up`' or '`hg up default`' or any other available commit you want to build
3. Clone the other needed repos (lmacros, ldosboot, instsect) from https://hg.pushbx.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. Refer to section 13.2 for the instsect help. The help can also 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).

3.2.2 How to prepare the test suite

The test suite (test/test.py) by default uses qemu. (dosemu2 tends to need more than 5 seconds to start while qemu manages in 2 seconds or less.)

If the debugger is run as a DOS application and qemu is used then a boot image containing a DOS kernel, shell, autoexec.bat, and quit program must be created. If the build option use_build_qimg is enabled then calls to mak.sh will create such an image. The script file makqimg.sh carries out this task.

If the debugger is run as a DOS application and dosemu2 is used then the DOS installed in dosemu is used. The -K and -E switches to dosemu2 are used to mount a host directory and execute the debugger.

If the debugger is bootloaded (in either qemu or dosemu2) then a boot image with only

the debugger executable and a startup boot script file must be created. If the build option use_build_bimg is enabled then calls to mak.sh will create such an image. The script file makbimg.sh carries out this task.

The test script creates symlinks to bin/ and tmp/qemutest/ and tmp/bdbgtest/ on its own. It can be executed from any directory, as it should find its files based on its own location. The test suite uses pseudoterminals, qemu or dosemu2, and the default Python unittest module.

Some tests may require having executed the script file test/scripts/mak.sh from within the test/scripts directory. When booting the debugger or using qemu, this must be run before makbimg.sh or makqimg.sh is run.

The DPMI tests currently require manual setup, with a directory test/dpmitest/ containing the dpmitest programs (for dosemu2) or a diskette image test/dpmi.img containing the programs as well as the HDPMI host executable (for qemu).

3.3 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.)

_DEBUG_COND

Only takes effect if _DEBUG option is also enabled. Allow to enable or disable debuggable mode within the same process. A ‘C’ is usually prepended to the program name. To indicate that the debuggable mode is enabled, the debugger's default command prompts are prepended by a tilde ‘~’.

The command-line switch /D+ can be used to start up in debuggable mode. /D- instead insures to start up in non-debuggable mode. The DCO6 flag 100h can be toggled subsequently to toggle debuggable mode.

_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 14.10.

_HISTORY

Enables the line editing history for raw terminal and serial input. Defaults to on. Size can be specified using _HISTORY_SIZE. Whether a separate segment is used can be controlled using the _HISTORY_SEPARATE_FIXED option. Defaults to an 8 KiB separate segment buffer.

_MEMREF_AMOUNT

Indicates number of memref structures to include. Default 4 (on). If enabled without a value, the default (4) is selected. When enabling this option, you most likely want to first rebuild the assembler and disassembler tables using the command `./mktables direction stackhinting`. (These mktables switches are now default enabled.) This allows for memrefs to indicate whether an explicit memory operand is a read or write (`direction`), as well as for stack accesses like `push`, `pop`, `call`, `retn` to be recognised in memrefs (`stackhinting`). Memrefs are initialised by disassembly. Memrefs can be accessed using the access variables like `READADR0`, `READLEN0`, etc. Refer to section 10.15. The access variables are written after an R command's register dump and disassembly (refer to section 9.29). Access variables can be accessed using special keywords behind the `IN` of a `VALUE x IN y` construct (refer to section 8.7).

Note that memrefs are not always exact. For instance, accesses by some instructions are not detected (eg `lgdt`, `sgdt`, `fsave`). Some instructions' accesses are not always correctly detected, such as `enter` with non-zero second operand, string instructions spanning segment boundaries, or instructions using `ss` after a write to `ss` that causes disassembly repetition. Some types of accesses are never detected either, such as GDT/LDT accesses to load descriptors. The stack access of software interrupt instructions is correctly detected only when tracing interrupts (Trace Mode set to 1, refer to section 9.41); if the interrupt call is proceeded past then like any proceeded-past function call it may use more stack space.

_SYMBOLIC

Enables the symbolic debugging support. This currently defaults to off. Documentation about the symbolic debugging support is still lacking.

Section 4: Getting started with the release

The stand-alone and FreeDOS release packages contain the following files:

In the `bin` or `BIN` directory:

`ldebugu.com`

Uncompressed bootable debugger, build without DPMI support

`ldebug.com`

Compressed bootable debugger, build without DPMI support

`ldebugxu.com`

Uncompressed bootable debugger, build with DPMI support

`ldebugx.com`

Compressed bootable debugger, build with DPMI support

`instsect.com`

Application to install boot sector loaders, with lDOS loaders that default to load `LDEBUG.COM` from a FAT12, FAT16, or FAT32 file system

The `tmp` or `SOURCE/LDEBUG/ldebug/tmp` directory contains subdirectories for each used compression method. For example, there is a subdirectory named `lz4`. These subdirectories contain the compressed executables `ldebug.com` and `ldebugx.com` built with the corresponding compression method.

NB: The default choice of compression method (LZMA-lzip) is chosen based purely on the smallest possible executable size. It may be unsuitable for use on low-end systems where it may take several minutes to decompress the application. In this case, the uncompressed executables may be used, or those compressed with another method (as found in the `tmp` subdirectories).

In the `doc` directory, or `DOC/LDEBUG`:

`ldebug.htm`

This manual in HTML, preferred form

`ldebug.txt`

Manual in plain text (FreeDOS package: with CR LF line endings)

`ldebug.pdf`

Manual in PDF

`fdbuild.txt`

FreeDOS package build instructions

`LDEBUG.LSM`

LSM file for lDebug FreeDOS package

In the root directory, or also `DOC/LDEBUG`:

`license.txt`

Full license texts for lDebug

In the `APPINFO` directory, only for FreeDOS package:

`LDEBUG.LSM`

LSM file for lDebug FreeDOS package

In the `lst` or `SOURCE/LDEBUG/ldebug/lst` directory:

`debug.lst`

Assembly listing corresponding to `ldebug.com` and `ldebugu.com`

`debug.map`

Assembly map corresponding to `ldebug.com` and `ldebugu.com`

`debugx.lst`

Assembly listing corresponding to `ldebugx.com` and `ldebugxu.com`

`debugx.map`

Assembly map corresponding to `ldebugx.com` and `ldebugxu.com`

Section 5: Invoking the debugger

5.1 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 9.18), 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 9.48). The filename is however `LDDEBUG.SLD` for DDebug builds.

Executing the `Q` command (section 9.26) makes the debugger uninstall itself then continue running whatever code the debuggee is in. Executing the `BOOT QUIT` command (section 14.10) 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.

5.2 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 command help page about invoking the debugger. Refer to section 13.1 for a copy of that help.

`/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.

`/S`

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

`/B`

Run a breakpoint within the debugger's initialisation.

`/V`

Enable/disable video screen swapping. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. Refer to section 9.44.

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 9.26) 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 9.42) 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.

5.3 Invoking the debugger as a device driver

The debugger's MZ .EXE style executable can also be loaded as a device driver. Loading as a device driver requires an MS-DOS version 5 level feature. Namely, the loader has to initialise and pass the pointer to the end of memory available to the device driver. (The debugger attempts to detect whether this pointer is passed and indicates enough memory, but it is unclear how well that works.)

Device drivers can be loaded from CONFIG.SYS using a `DEVICE=` directive. Other loaders such as DEVLOAD may work too. (DEVLOAD 3.25 specifically needs a patch to fix some problems keeping track of memory and to allow DEVLOAD to report more than 64 KiB of memory available to the device driver.)

DOS device loaders generally convert the device driver's command line to allcaps. To work around this, the debugger will interpret the exclamation mark in a special way: An exclamation mark indicates to convert the next letter to a small letter, if it is a capital letter. To pass a literal exclamation mark, double it.

All command line switches of the application mode are also accepted by the device mode debugger. In particular, `/C=` can be used to pass commands to execute.

The debugger will start up with debuggee client registers set up from the way they were passed by the device loader. CS:IP will point to a far return instruction in the debugger's entry segment. The stack will be preserved from what the device loader passed, too. That means running the debuggee allows to return control to DOS and have it finish installation of the debugger as a device. Subsequently, DOS and other device drivers and applications can be debugged, just like when resident in TSR mode.

The device mode debugger can terminate in two different modes. Both require a specific command letter appended to the Q command.

QD may be used if control did not return to the device loader yet. The debugger checks this condition by stashing away a copy of all regular registers to compare to their current values. This includes all GPRs, all segment registers, EIP, and EFL. Also, the debugger's device header

fields for pointing to the next device header are compared to FFFFh. If both match, it is assumed that we can still modify the request header passed by the device loader. This allows to report an error and set up an empty memory block to keep, so that the loader will know to discard the device.

QC may be used if control has returned to the device loader already and the debugger device has been installed into the system. It requires locating the device header in the chain of devices that starts with the NUL device in the DOS data segment. It also requires to find the memory block containing the debugger. It must be either a PSP-alike MCB (self-owned regular MCB containing exactly the debugger allocation) or an 'SD' (System Data) container MCB with one or more sub-MCBs (one of which contains exactly the debugger allocation). If these conditions are met, the debugger can be quit. It re-uses parts of the TSR application mode termination.

NOTE: Using QC currently assumes that no system file handles are left allocated to the placeholder character device that the debugger installs to keep itself resident. This device is currently called 'LDEBUG$$'. If this rule is not followed the system might crash.

5.4 Invoking the test suite

Use the test.py script in the test subdirectory. Use the -v switch to do verbose output. Specify test name patterns to use with -k, or omit to run all tests. The script uses the following environment variables:

build_name

Build name to use. Either debug (default), debugx, ddebug, or ddebugx.

test_booting

If set to a nonzero number, boot into the debugger. Otherwise, a DOS is loaded and the debugger is run as an application. Some tests are booting only, some other tests are non-booting only. The unsupported tests are skipped automatically.

test_initialise_commands

Commands to be executed by the test set up method right after establishing serial I/O. Semicolons are replaced by Carriage Returns.

DEFAULT_MACHINE

qemu or dosemu

DOSEMU

dosemu executable to use

QEMU

qemu executable to use

DEBUG

If set to a nonzero number, dump all serial I/O and all debugging messages.

Section 6: Interface Reference

6.1 Interface Output

The debugger provides a line-based text interface. The interface is written to DOS standard output by default. If InDOS mode is entered or the debugger is bootloaded then the interface is written to the terminal using interrupt 10h. Serial I/O can be enabled to write the interface to the serial port.

6.2 Interface Input

The default command prompt indicates that a command may be entered. It is a dash '-' by default, or a hash sign '#' when DebugX is in Protected Mode. An exclamation point '!' is prepended by a DOS application debugger (not bootloaded) while DOS's InDOS flag is set. A tilde '~' is prepended for DDebug.

If DOS command line input is done as raw input (eg if DCO option 800h is set) or the input is from a raw (ROM-BIOS) terminal, or from a serial port, then the line editing history is enabled. Prior commands may be recalled using the Up arrow key. The Down arrow key may also be used to reverse the recall. As soon as any prior or new line is edited the history recall is disabled.

Long command output may be paged. In that case, once a screenful has been displayed, a '[more]' prompt is displayed to pause the output. After pressing any key the output is continued. If Control-C is pressed, the current command is aborted.

6.3 Enabling serial I/O

Refer to section 10.8 for the serial configuration variables. Setting the DCO flag 4000h enables serial I/O. Upon enabling serial I/O a prompt is sent to the serial port. This prompt looks like the following example:

```
lDebug connected to serial port. Enter KEEP to confirm.
=
```

(The name of the debugger is modified to indicate DebugX, DDebug, or DDebugX. The prompt indicator is '~= ' for DDebug.) If the keep prompt is successfully displayed by the serial terminal and is responded to with the requested 'KEEP' keyword then serial I/O is established.

If the confirmation does not occur after a timeout then serial I/O is disabled again. The timeout defaults to about 15 seconds. In this case the debugger itself clears the DCO flag 4000h.

If the DCO flag 4000h is cleared then serial I/O is disabled.

6.4 Register dumping

The R command (refer to section 9.29) without any parameters dumps the current register values.

Then it disassembles a single instruction, or occasionally more than one. The register dump looks like this by default:

```
-r
AX=0000 BX=0001 CX=58A0 DX=0000 SP=0800 BP=0000 SI=0000 DI=0000
DS=1BEC ES=1BEC SS=35A9 CS=1BEC IP=0140 NV UP EI PL ZR NA PE NC
1BEC:0140 8CC8              mov     ax, cs
-
```

If the 'RX' command was used to switch on 32-bit register dumping, then the register dump looks like this:

```
-r
EAX=00000000 EBX=00000001 ECX=000058A0 EDX=00000000 ESP=00000800 EBP=0000
ESI=00000000 EDI=00000000 NV UP EI PL ZR NA PE NC
DS=1BEC ES=1BEC SS=35A9 CS=1BEC FS=0000 GS=0000 EIP=00000140
1BEC:0140 8CC8              mov     ax, cs
-
```

The RE command (section 9.29.1) runs the RE buffer commands. The default RE buffer content is a single '@R' command. After running the program being debugged, usually the RE buffer commands are also being run. This includes a step with the T, TP, or P commands. (Section 9.40, section 9.40.1, section 9.25.) It also includes a run with the G command. (Section 9.14.) Further, a permanent breakpoint which is configured as a pass point being passed also runs the RE buffer commands. (Section 9.5.)

Setting the flags 10000 or 40000 in the DCO3 variable enables register change highlighting. When output is written to DOS standard output or to a serial port then ANSI escape sequences are used to highlight. Specifically, '\x1B[7m' is used to reverse video and then '\x1B[m' to reset the colours.

For DOS standard output it may be needed to install an ANSI escape sequence parser.

For serial I/O the terminal connected to the debugger is expected to handle the escape sequences.

If the output is to a terminal using interrupt 10h and DCO3 flag 20000 is clear and the terminal is detected as functional then highlighting is done using interrupt 10h video attributes.

The functionality check is done by calling interrupt 10h service 03h. If the indicated current column is nonzero then the terminal is considered functional. (Current dosemu2 in `-dumb` terminal mode is detected as not being functional.)

If this check fails or the DCO3 flag 20000 is set then escape sequences are written using interrupt 10h.

6.5 Memory dumping

Another basic command is the D command (section 9.8). It is used to dump memory contents. For example, to dump part of a program:

```
-d
1BEC:0140  8C C8 31 DB 05 70 14 50-53 CB 70 03 91 67 BC 45 ..1..p.PS.p..g
1BEC:0150  3F 10 C1 6F F9 70 BA 22-7C 71 C3 72 0A 81 0A 81 ?..o.p."|q.r..
1BEC:0160  47 74 68 76 6C 77 32 72-A7 2F BD 78 4B 16 9F 7B Gthvlw2r./.xK.
```

```
1BEC:0170  C9 2B 09 37 0A 81 81 7D-E2 7E AC A0 00 00 00 00 .+.7...}.~....
1BEC:0180  10 49 00 00 0F 00 00 00-00 00 00 00 10 49 00 00 .I...........I
1BEC:0190  0F 00 00 00 F8 30 80 00-00 00 00 00 80 00 00 00 .....0........
1BEC:01A0  07 00 00 00 07 00 00 00-00 00 00 00 00 00 00 00 ................
1BEC:01B0  00 00 00 00 97 65 00 00-00 00 00 00 00 00 00 00 .....e........
-
```

Or, to dump the stack as words:

```
-dw ss:sp
header     0    2    4    6    8    A    C    E    0123456789ABCDEF
35A9:0800  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0810  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0820  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0830  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0840  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0850  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0860  0000 0000 0000 0000-0000 0000 0000 0000 ................
35A9:0870  0000 0000 0000 0000-0000 0000 0000 0000 ................
-
```

6.6 Disassembly

The U command is used to disassemble one or several instructions. Example:

```
-u
305C:0000 8CD0              mov     ax, ss
305C:0002 8CDA              mov     dx, ds
305C:0004 29D0              sub     ax, dx
305C:0006 31D2              xor     dx, dx
305C:0008 B90400            mov     cx, 0004
305C:000B D1E0              shl     ax, 1
305C:000D D1D2              rcl     dx, 1
305C:000F E2FA              loop    000B
305C:0011 50                push    ax
305C:0012 01E0              add     ax, sp
305C:0014 83D200            adc     dx, +00
305C:0017 83C00F            add     ax, +0F
305C:001A 83D200            adc     dx, +00
305C:001D 24F0              and     al, F0
305C:001F 83FA01            cmp     dx, +01
-
```

6.7 Help

The online help can be accessed using the '?' command. Refer to section 14 for copies of the online help.

Section 7: Parameter Reference

7.1 Number

Plain numbers are evaluated as expressions. Refer to section 8. 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'.

7.2 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. If the first number is specified as a pointer type using the type keyword '`POINTER`' then its upper 16 bits are taken as segment and its lower 16 bits are taken as 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.

If a segment or pointer type expression are prefixed by a dollar sign '`$`' then the specified segment is always taken as a Real/Virtual 86 Mode segment, even if DebugX is in Protected Mode. Otherwise, in Protected Mode a segmented address refers to a selector.

Address parameters are used by a lot of commands.

7.3 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 9.8), but they can be used for any command that accepts a range.

Range parameters are used by a lot of commands.

7.4 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. A quoted string can be delimited by single quotes ' or double quotes ". If the used delimiter quote mark occurs twice back to back while reading the quoted string, this is taken as an escape to include the delimiter mark itself as a byte of the string. List parameters are used by the E, F, and S commands. Refer to section 9.12, section 9.13, and section 9.38.

7.5 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 9.38 and section 9.13.

7.6 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.

7.7 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 9.5.

7.8 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 9.10 and section 14.10.

7.9 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 9.5 and section 9.14.

7.10 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 9.15 and section 9.48.

7.11 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 9.17 and section 9.24.

7.12 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 9.20 and section 9.46. The N and Y commands (section 9.23 and section 9.48) also accept drive parameters, but only as part of their filenames. These must be in the drive letter followed by colon format.

7.13 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 9.20 and section 9.46. Some BOOT commands also use sector numbers, refer to section 14.10.

7.14 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 9.18, section 9.25, section 9.40, section 9.5.3, section 9.5.1. 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.

7.15 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 9.29.

7.16 Command

Command is a special parameter type that is used only by the RE.APPEND, RE.REPLACE, RC.APPEND, and RC.REPLACE commands (section 9.29.2 and section 9.29.4). It is read verbatim and entered into the RE or RC 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 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 buffer.

7.17 ID

ID is a special parameter type that is used only by the BP and BI commands (section 9.5.1 and section 9.5.2). 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 8: Expression Reference

8.1 Literals

Literals consist of one or more digits. A literal must start with a digit or hash sign '#'. Embedded underscores '_' are skipped. Literals must not overflow 4 giga binary minus 1, that is FFFF_FFFFh.

The default base for literals is sixteen (hexadecimal). A hash sign '#' indicates a base change. If nothing preceeds the hash sign the base is changed to ten (decimal). Otherwise, the number before the hash sign is read in the prior base and taken as the base to change to. The base must be between 2 and 36. Multiple hash signs are allowed in the same literal.

8.2 String literals

String literals consist of up to 4 bytes. The bytes are specified starting with a hash sign '#' followed by a single-quote mark `'` or double-quote mark `"`. The same quote mark is used to end the string literal. If the delimiter quote mark occurs twice back to back while reading the string literal, that is handled as an escape to include the delimiter mark itself as a byte. Strings are read in a little-endian order, same as NASM does. That is, the first byte of a multi-byte string is read into the lowest byte of the numeric value. This matches the order obtained by writing the string to memory and reading it as a word, 3byte, or dword.

8.3 Variables

A variable consists of a variable name, possibly followed by parentheses with an index expression. Variable names are capitalisation insensitive. Variables differ in size, there are variables consisting of 8, 16, 24, or 32 bits. Variables can be written to using the R command. Some variables are read-only. A few variables allow writing some but not all bits.

8.4 Indirection

Indirection is indicated by square brackets. Within the brackets an address is parsed, defaulting to `ds` as the segment. The size of the indirect access can be specified with a type specifier before the brackets. The usual types are `BYTE`, `WORD`, `3BYTE`, and `DWORD`. Like variables, indirection terms can be written to using the R command.

8.5 Parentheses

Parentheses can be used to force a different order of operations.

8.6 `LINEAR` keyword

A keyword reading `LINEAR` introduces an address to parse. The address defaults to `ds` as the

segment. The address may be separated from subsequent text with a comma. If the expression is to be separated from a subsequent element using a comma after a LINEAR address then two commas are needed. Depending on the segmentation scheme of the current mode the segmented address is converted into a linear address. If DebugX is in Protected Mode and the segment base cannot be determined the expression is rejected as an error.

8.7 VALUE IN construct

A keyword reading VALUE starts a VALUE IN construct. Between the VALUE and subsequent IN keyword there is a single value expression, or a range of the form FROM expression TO expression or FROM expression LENGTH expression. Next follows the IN keyword. After this, there is a list of match ranges. A match range is either a single value expression, or a range of the form FROM expression TO expression or FROM expression LENGTH expression. After each match range a comma indicates another match range follows.

In a FROM TO specification the first expression has to evaluate to unsigned below-or-equal the second expression. In a FROM LENGTH specification the length must be nonzero. If these conditions are not met then the value or match range in question is always considered as not matching.

The entire VALUE IN construct evaluates to how many of the match ranges match the value range. The construct only evaluates to zero if no matches occurred. A nonzero value indicates that at least one match occurred.

8.7.1 VALUE IN construct keywords

Instead of a value or match range as specified here, the keyword EXECUTING may be specified. This expands to the following input:

```
FROM LINEAR cs:eip LENGTH abo - eip
```

If the _MEMREF_AMOUNT build option is enabled and paired with the direction and stackhinting switches to mktables then additional keywords are available for VALUE IN match ranges. That is, these keywords must be specified behind the IN and cannot be specified between the VALUE and IN.

These keywords are as follows:

READING

> Expands to a comma-separated list of FROM readadr0 LENGTH readlen0 constructs, for every read access variable pair (refer to section 10.15).

WRITING

> Expands to a comma-separated list of FROM writadr0 LENGTH writlen0 constructs, for every write access variable pair (refer to section 10.15).

ACCESSING

> Expands to READING, WRITING, EXECUTING.

8.8 Conditional ?? : : construct

The ternary conditional operator takes three operands. It is the only ternary operator.

The first operand, the condition, is specified before the ?? keyword. Note that the ?? keyword must be terminated by a blank or an opening square bracket or round parenthesis.

The second operand is specified between the ?? keyword and the : : keyword. Its value is used as the construct's return value if the condition is true.

The third operand is specified after the : : keyword. Its value is used as the construct's return value if the condition is false.

The conditional operator can be nested freely. The conditional operator must not be combined into the R command's assignment operator as in ??:=. The third operand may be separated from subsequent text with a comma. If the expression is to be separated from a subsequent element using a comma after a conditional's third operand then two commas are needed.

Section 9: Command Reference

9.1 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 10.3.)

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 10.3.)

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 10.3.)

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.

9.2 ? 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 14.

9.3 : prefix - GOTO label

A leading colon indicates a destination label for GOTO, see section 9.15.

9.4 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.

9.5 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.

9.5.1 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 9.5.4 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.

9.5.2 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.

9.5.3 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.

9.5.4 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.)

9.5.5 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.

9.5.6 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.

9.5.7 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.

9.5.8 BE command - Enable breakpoint

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

Like BD, but enables breakpoints.

9.5.9 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.

9.5.10 BS command - Swap breakpoint

```
swap            BS index1 index2
```

This command is provided to allow re-ordering existing breakpoints. It takes two indices both of which must refer to valid breakpoints. However, it is allowed to specify the index of an unused breakpoint for either of the parameters (or even both). All data associated with the two breakpoints is swapped.

9.5.11 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
-
```

9.6 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.)
-
```

9.7 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.

9.8 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.

9.9 DI command - Dump Interrupts

```
dump interrupts DI[R][M][L] 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. In 86 Mode, an interrupt chain is displayed if more than one entrypoint is reachable from the topmost handler. To make the next handler reachable, a handler must match one of several header / entry formats:

- IBM Interrupt Sharing Protocol (IISP) header (fully standard, with 10EBh entrypoint and EBh jump to hardware reset - this matches what Ralf Brown's AMIS programs recognise)
- Non-standard IISP header
- iHPFS-style uninstalled IISP header (EA90h entrypoint)
- FreeDOS kernel relocation (near call followed by far jump immediate)
- Just a far jump immediate

If the R is specified (directly after DI) then 86 Mode handlers are dumped even if in PM.

If the M is specified then MCB names are displayed.

If the L is specified then AMIS interrupt lists are queried for the interrupt number being dumped. This is so that the involved multiplex numbers and interrupt list indices can be displayed, and also so that hidden chains can be dumped. This means chains that are not reachable from the topmost IVT handler, but are found through the AMIS "Determine Chained Interrupts" call (either 03h pointer or 04h list return). The list index is displayed as FFFFh if the handler was found with 03h pointer return. Otherwise it indicates how many list entries precede the found handler's entry. For example, '`list:0000h`' means that the first list entry matched, and '`list:0001h`' means that the second list entry matched.

Specifying the L makes the debugger use its auxiliary buffer. That means the DIL command cannot be used from the RE buffer if either a T/TP/P WHILE condition is used, or the T/TP/P silent buffer is used, or both. In addition, note that with the default buffer size, no more than about a 1000 handlers can be handled. (The actual limit may be as low as 500 handlers if a lot of hidden chains occur.) If the limit is exceeded then the DIL command will display an error. The same error can also occur if the chain loops, or references a single handler from more than one other handler, or a single handler is listed by more than one multiplexer.

9.10 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.

9.11 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.

9.12 E command - Enter memory

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

The E command is used to enter values into memory. If the list is specified, its contents are written to the address specified. Otherwise, the interactive enter mode starts at the address specified.

In the interactive enter mode, the segmented address is displayed, and then the current byte value (2 hexadecimal digits) found at that address yet. Following the value a dot is displayed. For example:

```
-e 100
1FFE:0100  C3.
```

At this point the debugger accepts several different inputs:

- One or two hexadecimal digits: To enter a new value to be written at this address
- A blank: To write the new value (if any) and proceed to the next byte
- A minus: To write the new value (if any) and proceed to the prior byte
- Carriage Return, Line Feed, or a period: To write the new value (if any) and quit interactive enter mode
- Backspace: To delete the most recently entered digit of a candidate new value
- All other inputs are ignored

After entering a blank, the debugger will either display the next byte's current value in the same line or start a new line with the current segmented address and then the current byte value. A new line is started if the current offset is divisible by 8. For example, after entering 8 blanks:

```
-e 100
1FFE:0100  C3.     CC.     CC.     CC.     CC.     CC.     CC.     CC.
1FFE:0108  CC.
```

After entering a minus, the minus is displayed on the current line and then (always) a new line is started to display the new segmented address (with its offset decremented). For example, entering a new value (‘A0’), then a blank, then a minus, and then another new value (‘A1’), then a CR:

```
-e 100
1FFE:0100  C3.A0   CC.-
1FFE:0100  A0.A1
-
```

9.13 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.

9.14 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.

9.15 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.

9.16 H command - Hexadecimal add/subtract values

```
hex add/sub     H value1 [value2 [...]]
base display    H BASE=number [GROUP=number] [WIDTH=number] value
```

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.

If the form with the `BASE` keyword is given then only one number is displayed. The specified base may be between 2 and 36, inclusive. If the `GROUP` keyword is also used then digits are grouped. The group separator is the underscore, '_'. The grouping number must be below or equal 32 (20h). The default grouping is none, same as `GROUP=0`. If the `WIDTH` keyword is also used then at least that many digits are displayed. The width must be below or equal 32 (20h). The default width is one digit, same as `WIDTH=0` or `WIDTH=1`.

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)
-h base=2 group=8 AA55
10101010_01010101
-h base=2 group=4 width=#16 #1234
0000_0100_1101_0010
-h base=#10 group=3 400*400
1_048_576
-h base=3 group=3 FFFF_FFFF
102_002_022_201_221_111_210
-
```

9.17 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.

9.18 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 9.15).

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 9.48). 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.

9.19 L command - Load Program

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

9.20 L command - Load Sectors

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

9.21 M command - Move memory

```
move            M range address
```

9.22 M command - Set Machine mode

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

An M command without parameters, with a single '`?`' parameter, with an '`NC`' parameter, or a single expression parameter is a get or set machine mode command.

The machine mode is used by the assembler and disassembler to show machine requirements exceeding the current machine.

A plain '`M`' or '`M ?`' command displays the current machine.

An '`M NC`' or '`M C0`' command sets the current coprocessor to absent.

An '`M C`' command sets the current coprocessor to present. It is set to the coprocessor type corresponding to the current machine.

An '`M C2`' command sets the current coprocessor to present, and the coprocessor type to 287. This command is only valid if the current machine is a 386.

An M command with an expression evaluating to 0 to 6 sets the current machine to the specified numeric value. It also sets the current coprocessor type corresponding to the specified numeric value. Coprocessor presence is not modified by this command however.

9.23 N command - Set program Name

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

9.24 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.

9.25 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.

As an exception, if a near immediate '`call`' (opcode E8h) is to be executed and its callee is a '`retf`' or '`iret`' instruction, then the '`call`' instruction is traced and not proceeded past. (This supports some relocation sequences.)

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.

9.26 Q command - Quit

```
quit            Q
```

9.27 QA command - Quit attached process

```
quit process    QA
```

The QA command tries to quit an attached process. It does this by resetting the current cs:eip, ss:esp, efl, and (only for DebugX) all segment registers. Then it runs interrupt 21h service 4C00h in the context of the current debuggee. Afterwards it reports on how the debugger regained control and whether the attached process terminated.

(If between the current debuggee's process and the debugger's process there is any process that is self-parented, or a breakpoint interrupt or trace interrupt is caused by the current process having terminated, then the attached process may be considered not terminated.)

The same underlying function is used by the program-loading L command and the default Q command (except if the debugger is running in TSR mode).

9.28 QB command - Quit and break

```
quit and break  QB
```

The QB command is composed of a Q command with a B flag. It indicates to the debugger to quit as usual, but to then run a breakpoint just before the debugger returns the control flow to either the OS, the application that executed the debugger, or (when resident as TSR, device driver, or bootloaded) the current debuggee.

When successful, this instance of the debugger has already uninstalled all its interrupt hooks, so the breakpoint will run the interrupt 3 handler that was installed prior to the debugger having been installed.

9.29 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
-
```

9.29.1 RE command - Run register dump Extended

```
Run R extended  RE
```

The RE command runs the RE buffer commands. Refer to section 14.7.

9.29.2 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 14.7).

9.29.3 RC command - Run Command line buffer

```
Run Commandline RC
```

The RC command runs the command line buffer commands. This is similar to the RE command, except it uses a different buffer. Upon initialisation of the debugger the RC buffer is filled with the content of the `/C` switch (if any) in case the debugger is loaded as a DOS application, or else the contents of the kernel command line (if any) or the default kernel command line contents. Then the equivalent to an RC command is run.

Command line buffer commands are displayed with a prompt consisting of an ampersand `&` or, for DDebug, a tilde followed by an ampersand `~&`. When both RE and RC are running out of their respective buffers, the RE buffer contents take precedence.

9.29.4 RC buffer commands

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

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

RC.APPEND appends the following commands to the command line buffer.

RC.REPLACE replaces the command line buffer with the following commands.

9.30 RM command - Display MMX Registers

```
MMX register    RM
```

9.31 RN command - Display FPU Registers

```
FPU register    RN
```

9.32 RX command - Toggle 386 Register Extensions display

```
toggle 386 regs RX
```

9.33 RV command - Show sundry variables

This command shows the first 16 user-defined variables (refer to section 10.12), the current options variables DCO (that is DCO1), DCS, DAO, DAS, the internal flags DIF (that is DIF1), as well as the debugger process segment (DPR), the debugger parent return address (DPI), and the debugger parent process (DPP). lDebugX also shows the debugger process selector (DPS), which is zero in 86 Mode and a selector value in Protected Mode. (All of these variables can be

queried manually, the RV command lists them merely for convenience.)

Additionally, in the last line the RV command displays the current debuggee's mode. This is either Real 86 Mode, Virtual 86 Mode, or (lDebugX only) Protected Mode with either a 16-bit CS or a 32-bit CS.

9.34 RVV command - Show nonzero user-defined variables

This command shows all user-defined variables (refer to section 10.12) that are not currently zero. Variables are always shown four to a line, so a single non-zero variable will additionally show up to 3 variables that are currently zero.

9.35 RVM command - Show debugger segments

This command shows various segments (and, in Protected Mode, selectors) used by the debugger. It currently shows the following:

- Code segment
- Data segment
- Entry segment (same as data segment but with a code selector in PM)
- Auxbuff segment
- History segment

9.36 RVP command - Show process information

This command shows the debugger's mode as well as some client and debugger process addresses. The mode is one of:

- Boot loaded
- Device driver
- Application
- Application installed as TSR

The process addresses include:

PSP

Process Segment Prefix (always a 86M segment value)

Parent

Parent of the PSP (for the debugger the would-be parent for termination, however note that during normal operation the debugger is self-parented)

Parent Return Address

16:16 far pointer (a segmented 86M far address) of the process's interrupt 22h value, the entrypoint to return to the parent (again for the debugger this is the would-be PRA for termination, during normal operation the debugger sets up its actual PRA to return control to the debugger itself)

PSP Selector (only displayed for lDebugX)

A selector or segment value, appropriate for the current mode, to address the PSP

The process addresses can all be accessed individually too, using the following variables:

PSP

PSP (client), DPSP (debugger)

Parent

PARENT (client), DPARENT (debugger)

Parent Return Address

PRA (client), DPRA (debugger)

PSP Selector (always a segment if not lDebugX)

PSPSEL (client), DPSPSEL (debugger)

The DPARENT and DPRA variables read as all zeros when the debugger is loaded in bootloaded, device driver, or resident application (TSR) mode.

9.37 RVD command - Show device information

This command shows the device header (segmented 86M) far address as well as the size of the device's allocation, in paragraphs. If the debugger is not loaded in device mode then instead a message indicating this is displayed.

The two variables can be accessed individually, too. These are the DEVICEHEADER and DEVICESIZE variables. Both of them read as all zeros when the debugger is not loaded in device mode.

9.38 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.

9.39 SLEEP command

```
sleep           SLEEP count [SECONDS|TICKS]
```

The SLEEP command sleeps for the indicated length. The duration defaults to seconds. If the TICKS keyword is specified then the duration is taken to mean timer ticks. (A timer tick is about 1/18 seconds.) If the input is from DOS or serial I/O then Control-C from the input terminal may be used to cancel the sleep.

9.40 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.

9.40.1 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.

9.41 TM command - Show or set Trace Mode

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

9.42 TSR command - Enter TSR mode

```
enter TSR mode  TSR
```

9.43 U command - Disassemble

```
unassemble      U [range]
```

9.44 V command - Video screen swapping

```
view screen     V [ON|OFF [KEEP|NOKEEP]]
```

The V commands allow to enable or disable video screen swapping. When enabled, the debugger takes care that screen output of debuggee and debugger are strictly separated. This is useful to debug fullscreen text mode programs.

The screen will be swapped whenever the debuggee is run with a run command (T/TP/P/G), or when the plain V command is used. The plain V command is provided to watch the debuggee screen while the debugger is active. It ends upon the user entering any key to the debugger terminal.

Video screen swapping currently requires an XMS driver, and the debugger will allocate an XMS memory block of 32 KiB.

V OFF KEEP will disable video screen swapping but keep the current debugger screen contents. V OFF NOKEEP (and the default for V OFF if the keep flag has not been set) will instead return to the debuggee screen contents. When the Q command succeeds, it executes the equivalent of V OFF. That is it will use the current keep flag.

9.45 W command - Write Program

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

9.46 W command - Write Sectors

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

9.47 X commands - Expanded Memory (EMS) commands

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

9.48 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 14.10 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.

Opening a script file as DOS application only works while DOS is available (InDOS not set). Additionally, if during script file execution DOS becomes unavailable (InDOS is set) then the script file execution is paused. It is resumed once DOS becomes available again. (Control-C with a non-zero IOL variable may still be used to cancel script file execution. DOS is called to close affected handles only if DOS is available.)

9.49 Z commands - Symbolic debugging support

These commands are only supported if the _SYMBOLIC build option is enabled.

9.49.1 Z /S=size - Allocate, resize, or free symbol tables

The /S switch allows to change the symbol table allocation. The symbol tables may take up up to 256 KiB of 86 Mode memory (below 1024 KiB) or up to 2 MiB of XMS memory. XMS use implies an additional 65 KiB is allocated for padding and a transfer buffer.

XMS use can be forced by using a letter X behind the /S. 86 Mode memory use can be forced by using a letter R instead. The prior selection can be undone using an asterisk *, returning to the default behaviour. That means allocate XMS if available, and fall back to 86 Mode memory otherwise.

After the equals sign a size is to be specified. The size can be an immediate number or an expression, or the keyword MAX to use the maximum size. An expression must be surrounded by round parentheses. The size specifies the amount of kibibytes to allocate. The size may be zero, which signals to free all symbol tables. This deletes all symbols yet defined. Otherwise, new symbol tables are allocated. Existing symbols will be transferred from the old symbol tables, if there are any. It is an error to specify a symbol table size that is not large enough to hold all currently defined symbols, except for specifying a zero size.

Multiple /S switches can be specified within the same Z command. They are processed one by one, that is an error during parsing or execution of a subsequent switch will not make it so a prior switch is skipped.

9.49.2 Z STAT - Show symbol table statistics

This command shows statistics on the current symbol table sizes, including the amount of total, used, and free units. Each of the symbol main array, symbol hash array, and symbol string heap are listed.

9.49.3 Z ADD - Add a symbol

This command is used to add a new symbol. It can be followed by several parameters. These are:

SYMBOL= or S=

Name of the symbol, may be quoted

OFFSET= or O=

Offset of the symbol

LINEAR= or L=

Linear address of the symbol

FLAGS= or F=

Flags of the symbol

No keyword

Segmented address of the symbol to specify the linear address and offset

9.49.4 Z DEL - Delete a symbol

This command deletes a symbol. It can be followed by the symbol name to delete, or a RANGE keyword and an address range parameter, or an UNREFSTRING keyword. The latter is to clean up the symbol string heap by deleting entries that are no longer used.

9.49.5 Z COMMIT - Commit temporary symbols

Z ADD will batch up new symbols as temporary symbols. They are committed into the symbol tables upon several conditions, such as no more space for temporary symbols or execution of a command other than Z ADD or Z ABORT. The Z COMMIT command is for forcing the temporary symbols be committed. This should not usually be required.

9.49.6 Z ABORT - Discard temporary symbols

This command discards all temporary symbols batched by prior Z ADD commands if they were not yet committed. If the debugger responds to every command with the error message "Invalid symbol table data!" then something went wrong with the committing of temporary symbols. In this case the Z ABORT command may help to return the debugger to a usable state.

9.49.7 Z LIST - List symbols

9.49.8 Z MATCH - Match symbols

9.49.9 Z RELOC - Relocate symbols

Section 10: Variable Reference

10.1 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`

10.2 Options

10.2.1 DCO - Debugger Common Options

10.2.2 DCS - Debugger Common Startup options

10.2.3 DIF - Debugger Internal Flags

10.2.4 DAO - Debugger Assembly Options

10.2.5 DAS - Debugger Assembly Startup options

10.2.6 DPI - Debugger Parent Interrupt 22h

10.2.7 DPR - Debugger PRocess

10.2.8 DPP - Debugger Parent Process

10.2.9 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.)

10.3 Default step counts

PPC

Proceed command (section 9.25) default step count

TPC

Trace/Proceed command (section 9.40.1) default step count

TTC

Trace command (section 9.40) 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 9.1. If one of these is set to zero then it is an error to not specify a count explicitly for the corresponding command.

10.4 Limits

10.4.1 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.

10.4.2 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.

10.5 Return Codes

10.5.1 RC - Return Code

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

10.5.2 ERC - Error Return Code

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

10.6 Addresses

10.6.1 A address (AAS:AAO)

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

10.6.2 D address (ADS:ADO)

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

10.6.3 Address behind R disassembly (ABS:ABO)

10.6.4 U address (AUS:AUO)

Default address for the disassembler.

10.6.5 E address (AES:AEO)

Default address for memory entry.

10.6.6 DZ address (AZS:AZO)

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

10.6.7 D$ address (ACS:ACO)

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

10.6.8 D# address (APS:APO)

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

10.6.9 DW# address (AWS:AWO)

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

10.6.10 DX address (AXO)

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

10.7 I/O configuration

10.7.1 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.

10.7.2 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.

10.7.3 IOS - I/O Circular Keypress Buffer Start

Word. Default 0 or 1Eh. Indicates where the ROM-BIOS's circular keypress buffer starts. Value can be nonzero to force a particular offset in segment 40h. Value can be zero to force using the value at `word [40h:80h]`, using an extension not available on all systems.

On startup the debugger checks whether the extension values are valid. If they are then the default

of the IOS variable is left as zero. Otherwise, the default is set to 1Eh, which is the default buffer location.

This variable is used to check for Ctrl-C keypresses if the InDOS mode is on (either InDOS flag set, DCO flag 8 set, or in bootloaded mode) and serial I/O is not in use and the flag DCO3 2000_0000h is set. Setting this variable nonzero and equal to IOE disables Ctrl-C checking.

Modifying this variable should only be done while it is not in use. That means using DOS for input, using serial I/O for input, or clearing the DCO3 flag 2000_0000h. Modifying this variable and the IOE variable should be done together, so that they are valid together when in use.

10.7.4 IOE - I/O Circular Keypress Buffer End

Word. Default 0 or 3Eh. Indicates where the ROM-BIOS's circular keypress buffer ends. Value can be nonzero to force a particular offset in segment 40h. Value can be zero to force using the value at `word [40h:82h]`, using an extension not available on all systems.

Refer to IOS description above.

10.7.5 IOL - I/O Amount of Script Levels to Cancel

Word. Default 255. Indicates how many levels of script files and RE buffer execution to cancel when a Control-C input or critical DOS error is detected by the debugger. The effective value will be incremented by one if IOF flag 1 is set and RE buffer execution is in progress.

Zero indicates to only cancel the current command. One indicates to cancel the current command, plus the RE buffer execution if any, else up to one level of script file execution. Two indicates to cancel two levels of execution: either the RE buffer execution and one level of script file execution, or up to two levels of script file execution.

The debugger always cancels RE buffer execution first if it is in progress. Next, the innermost script file execution is cancelled, if any.

10.7.6 IOF - I/O Flags

Word. Default 1. Flags for I/O handling. Currently defined:

1

> Extra IOL level for RE buffer execution. If set, RE buffer execution being in progress increments the effective value of the IOL variable.

10.8 Serial configuration

10.8.1 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.

10.8.2 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.

10.8.3 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.

10.8.4 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.

10.8.5 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.

10.8.6 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.

10.8.7 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.

10.8.8 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.

10.8.9 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.

10.8.10 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.

10.9 _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.

10.9.1 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.

10.9.2 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.

10.9.3 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.

10.9.4 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.

10.10 _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.

10.10.1 MT0 - Mask Test 0

Doubleword. Default 0. Mask AA55_AA55h.

10.10.2 MT1 - Mask Test 1

Doubleword. Default 0011_0022h. Mask 00FF_00FFh.

10.11 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.

10.11.1 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.

10.12 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.

10.13 PSP variables

10.13.1 PSP - Process Segment Prefix

10.13.2 PPR - Process PaRent

10.13.3 PPI - Process Parent Interrupt 22h

10.14 SR variables - Search Results

10.14.1 SRC - Search Result Count

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

10.14.2 SRS - Search Result Segment

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

10.14.3 SRO - Search Result Offset

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.

10.15 Access variables

These variables can be left out of the build. The build option `_MEMREF_AMOUNT` must be enabled to include them.

10.15.1 READADR

Doubleword. Read only. READADR0 to READADR3 each specify one variable. (Amount of READADR variables can be configured at build time with the option _ACCESS_VARIABLES_AMOUNT, which defaults to 4.) Linear addresses of string, stack, or explicit memory operand reads. Initialised by the R command. Unused variables are reset to zero by the R command. It is valid to refer to any READADR variable using an index expression. Index expression means that the variable name (`READADR`) is immediately followed by an opening parenthesis, followed by a numeric expression which evaluates to a number below 4.

10.15.2 READLEN

Doubleword. Read only. READLEN0 to READLEN3 each specify one variable. Length of string, stack, or explicit memory operand reads. Initialised by the R command. Unused variables are reset to zero by the R command. It is valid to refer to any READLEN variable using an index expression.

10.15.3 WRITADR

Doubleword. Read only. WRITADR0 to WRITADR3 each specify one variable. Linear addresses of string, stack, or explicit memory operand writes. Initialised by the R command. Unused variables are reset to zero by the R command. It is valid to refer to any WRITADR variable using an index expression.

10.15.4 WRITLEN

Doubleword. Read only. WRITLEN0 to WRITLEN3 each specify one variable. Length of string, stack, or explicit memory operand writes. Initialised by the R command. Unused variables are reset to zero by the R command. It is valid to refer to any WRITLEN variable using an index expression.

10.16 Machine type variables

MMT - Maximum Machine Type encountered

> Set whenever the disassembler encounters an instruction requiring a machine type that is higher than this variable's current value. Writable.

MACHX86 - Machine type for assembler and disassembler

> Current machine type to use for assembler and disassembler. Read-only, use M commands to modify.

MACHX87 - Coprocessor encoded machine type

> Contains valid argument to M command: C0h if no coprocessor, 0Ch if coprocessor matching machine, C2h if machine is a 386 with a 287 coprocessor. Read-only, use M commands to modify.

10.17 LFSR variables

These variables provide access to a simple LFSR (Linear Feedback Shift Register). The default taps are chosen so that a full-range 32-bit LFSR is in use. That means there are 4 giga binary steps, minus one, and all possible 32-bit values are in use except for the all zeros value. A step of the LFSR is done by shifting the old value to the right once. If the bit shifted out is a 1, then the new value is obtained by applying the LFSR taps as a XOR mask to the shift result. If the bit shifted out is a 0, then the new value is simply the shift result.

LFSR - Forward LFSR variable

> Whenever this variable is read, it first executes an LFSR step from the variable's prior value. What is actually read is the new value after the step. This variable is initialised to the constant 2 on startup of the debugger. That means that with the default taps, the first read will return 1, the second 8020_0003h, etc.

LFSRTAP - Taps to use for the LFSR

> This variable determines the tap bits to use for the LFSR. The default is 8020_0003h, leading to a full-range 32-bit LFSR. Different values may be chosen. The highest bit of the taps value determines how wide the forward LFSR is.

RLFSR - Reverse LFSR variable

> Similar to the forward LFSR variable, except it runs backwards. This also uses the LFSRTAP variable, however the taps are shifted to the left once, and the least-significant bit is set to 1. In addition, the RLFSRTOP variable is used to get the check mask, by shifting left the constant 1 by RLFSRTOP binary digits places. The check mask is used to determine whether to XOR mask with the taps or not. The check mask also indicates what bit to clear in the taps in order to create the reverse taps.

RLFSRTOP - Reverse LFSR top bit count

This variable indicates what bit to check in order to determine whether the reverse LFSR should tap or not. It also indicates what bit to clear in the creation of the reverse taps. Its default is 1Fh (31), which lends itself to a 32-bit taps value. Setting this to a number higher than 1Fh (31) is invalid, and may be subject to behaviour as yet undetermined.

Section 11: Interrupt Reference

11.1 Mandatory interrupt hooks

- Interrupt 0 - Divide error
- Interrupt 1 - Trace
- Interrupt 3 - Breakpoint
- Interrupt 6 - Invalid opcode
- Interrupt 18h - Diskless boot hook
- Interrupt 19h - Boot load

These interrupts are always hooked by the debugger. For the non-_DEBUG builds they are hooked during initialisation and the debugger attempts to unhook them when quitting. The highest 8 bits of the dword variable DCO4 control whether they are unhooked only if reachable (bits in DCO4 zero), or forcibly so if not reachable (bits in DCO4 ones). If not forcibly unhooking and an interrupt handler is not reachable then the Q command fails.

For DDebug, these interrupts are hooked within the `run` function and unhooked before the `run` function returns. This unhooking in DDebug is always forcible; that is, if not reachable then the interrupts are unhooked by simply updating the IVT entries with whatever handlers are stored as the next vectors in DDebug's entrypoints.

11.2 Serial interrupt

This interrupt hook is optional. Setting the DCO flag 4000h (enable serial I/O) instructs the debugger to set up this interrupt hook. Clearing the flag or using the `Q` command instructs the debugger to unhook its handler. The DCO4 flag 1_0000h controls whether the interrupt unhooking is forcible (flag set) or not (flag clear).

The exact interrupt number used as serial interrupt depends on the DSPVI variable at the point in time at which serial I/O is enabled. The default is interrupt 0Bh, corresponding to COM2.

11.3 Interrupt 2Fh - Multiplex (DPMI entrypoint)

This interrupt is only hooked by DebugX. This interrupt hook is optional. Setting the DCO4 flag 2 instructs the debugger to set up this interrupt hook. The debugger tries to hook this interrupt if it runs application code in Real or Virtual 86 Mode. Clearing the flag, entering Protected Mode, or using the `Q` command instructs the debugger to unhook its handler. The DCO4 flag 2_0000h controls whether the interrupt unhooking is forcible (flag set) or not (flag clear).

This interrupt is hooked to intercept calls to function 1687h, used to detect the DPMI entrypoint. DebugX attempts to hook this service to return its own entrypoint to the caller. The hook may

fail if the DPMI host handles interrupt 2Fh calls before chaining to the 86 Mode handler chain. (MS Windows 4.x and older dosemu are reported to do this.)

11.4 Interrupt 8 - Timer

This interrupt hook is optional. Setting the DCO4 flag 4 instructs the debugger to set up this interrupt hook. Clearing the flag or using the `Q` command instructs the debugger to unhook its handler. The DCO4 flag 4_0000h controls whether the interrupt unhooking is forcible (flag set) or not (flag clear).

This interrupt is used to detect the double Control-C via serial I/O condition. If the serial I/O handler of the debugger receives two Control-C keypresses while the debugger is busy running an application then the interrupt 8 hook will interrupt the run.

This interrupt is also used to detect the Control pressed for 5 seconds condition. Similarly to the serial I/O double Control-C condition, this will make the debugger interrupt the current run.

11.5 Interrupt 2Dh - Alternate Multiplex Interrupt

This interrupt hook is optional. Setting the DCO4 flag 8 instructs the debugger to set up this interrupt hook. Clearing the flag or using the `Q` command instructs the debugger to unhook its handler. The DCO4 flag 8_0000h controls whether the interrupt unhooking is forcible (flag set) or not (flag clear).

This interrupt allows other programs to detect the debugger in the AMIS interface. The vendor string is '`ecm`' and the product string '`lDebug`'. The description string contains the same display name and version as the command line help. There are two real uses of this. First, the AMIS function 4, which will return the list of interrupt entrypoints of the debugger. Second, lDebug's private AMIS function 30h. It is described in the next section.

This interrupt hook only succeeds if the current handler is valid. That is, an offset not equal to FFFFh and a segment not equal to zero. Another condition is that the debugger needs to detect an unused AMIS multiplex number to allocate. This is done automatically when hooking the interrupt. If either condition fails then a message is displayed and the debugger clears the DCO4 flag 8 on its own.

The TRYAMISNUM variable is a writable byte variable. It defaults to 0. Its content is tried first when searching a free multiplex number. After that the debugger currently will search starting from number 0 up to 255.

The AMISNUM variable is a read-only byte variable. It contains the actually used multiplex number while the DIF4 flag 8 is set. Otherwise its content is not used and likely stale.

11.5.1 AMIS private function 30h - Update IISP Header

This function is provided for use by our programs that use AMIS multiplexers and interrupt handler entrypoints with IISP headers. All TSRs (including RxANSI, lClock, SEEKEXT, KEEPHOOK, FDAPM, FreeDOS SHARE) and SHUFHOOK use this function. (Note that the debugger itself does not yet use this function.)

```
lDebug - Update IISP Header
INP:    al = 30h
        ds:si -> source IISP header (or pseudo header)
        es:di -> destination IISP header
```

```
OUT:    al = FFh to indicate suppported,
         si and di both incremented by 6
         destination's ieNext field updated from source
        al != FFh if not supported,
         si and di unchanged
CHG:    -
REM:    This function is intended to aid in debugging
         handler re-ordering, removal, or insertion.
         The 32-bit far pointer needs to be updated
         as atomically as possible to avoid using
         an incorrect pointer.
        Test case: Run a program such as our TSRs'
         uninstaller or SHUFHOOK and step through it
         with "tp fffff" when operating on something
         crucial such as interrupt 21h. Without this
         function the machine will crash!
        To enable this function to be called, enter
         the command "r dco4 or= 8" first (install our
         AMIS multiplexer handler).
        Other workaround: Use SILENT for TP and disable
         DCO3 flag 4000_0000 (do not call int 21.0B to
         check for Ctrl-C status).
        Yet another workaround: Set flag DCO 8 (enable
         fake InDOS mode, avoid calling int 21h).
REM:    The source may be a pseudo IISP header. In this
         case the ieEntry field should hold 0FEEBh
         (jmp short $) and the ieSignature field
         should indicate the source, eg "VT" for the IVT
         or "NH" for inserting a New Handler.
```

Section 12: Service Reference

These are the services called by the debugger.

12.1 Interrupt 10h

Used for output while InDOS, DCO flag 8 set, or bootloaded.

Function 02h

Set cursor position (only used if highlighting)

Function 03h

Get cursor position (only used if highlighting, indicates to highlight to int 10h if supported)

Function 08h

Get video attribute (only used if highlighting)

Function 09h

Set video attribute (only used if highlighting)

Function 0Eh

Teletype output

12.2 Interrupt 16h

Used for input while InDOS, DCO flag 8 set, or bootloaded.

Function 00h

Read keypress (wait until keypress available)

Function 01h

Read keypress (return if no keypress available)

12.3 Interrupt 2Fh

Function 1680h

Idle (Release timeslice to multitasker)

Function 1687h

Get DPMI entrypoint (used and hooked by lDebugX)

Function 4A06h

RPL adjust base memory size (called by booted debugger if RPL signature present)

12.4 Interrupt 12h

Called by booted debugger to determine base memory size.

12.5 Protected Mode Interrupt 31h

Used by lDebugX while in Protected Mode.

Function 0000h

Function 0002h

Get selector from segment

Function 0003h

Function 0006h

Get segment base

Function 0007h

Set segment base

Function 0008h

Function 0009h

Function 000Ah

Function 000Bh

Get descriptor

Function 000Ch

Set descriptor

Function 0200h

Get 86M interrupt vector

Function 0201h

Set 86M interrupt vector

Function 0202h

Get PM exception vector

Function 0203h

Set PM exception vector

Function 0204h

Get PM interrupt vector

Function 0205h

Set PM interrupt vector

Function 0300h

Call Real/Virtual 86 Mode interrupt

Function 0305h

Get raw mode switch save state addresses

Function 0306h

Get raw mode switch addresses

Function 0900h

Disable Virtual Interrupt Flag

Function 0901h

Enable Virtual Interrupt Flag

Function 0902h

Get Virtual Interrupt Flag

12.6 Protected Mode Interrupt 2Fh

Function 1680h

Idle (Release timeslice to multitasker)

Function 168Ah

Determine whether DOS extender is available.

12.7 Protected Mode Interrupt 21h

Function 7305h

Read/write sectors from/to DOS drive. Used to implement L and W command.

Function 4Ch

Terminate DPMI client and process

12.8 Protected Mode Interrupt 25h

Read sectors from DOS drive. Used to implement L command.

12.9 Protected Mode Interrupt 26h

Write sectors to DOS drive. Used to implement W command.

12.10 Interrupt E6h

Function bx = 0, ax = -1

Used by booted debugger to implement BOOT QUIT command when running in dosemu2.

12.11 Interrupt 15h

Function 87h

Used by DX command to read memory.

Function 5301h, 530Eh, 5307h

Used by booted debugger to implement BOOT QUIT command when running in qemu.

12.12 Interrupt 13h

Used by the booted debugger to load scripts or kernel executables.

Function 00h

Reset disk system

Function 02h

Read sector with CHS addressing

Function 03h

Write sector with CHS addressing

Function 08h

Query CHS geometry

Function 41h

Detect LBA extensions support

Function 42h

Read sector with LBA

Function 43h

Write sector with LBA

12.13 Interrupt 19h

Boot load. Used if booting the debugger fails.

12.14 Interrupt 2Dh

Used to access Alternate Multiplex Interrupt Specification TSRs. Can be used while bootloaded too.

Function 00h

Installation check. Determines whether an AMIS number is in use.

Function 04h

Determine chained interrupts. Determines interrupt entrypoints.

12.15 Interrupt 25h

Read sectors from DOS drive. Used to implement L command. Only used if the debugger is loaded as a DOS application.

12.16 Interrupt 26h

Write sectors to DOS drive. Used to implement W command. Only used if the debugger is loaded as a DOS application.

12.17 Interrupt 21h

DOS services. Only used while not InDOS. (Only used if the debugger is loaded as a DOS application.)

Function 08h

Get standard input keypress

Function 0Ah

Line buffered standard input

Function 0Bh

Check standard input available / Check Control-C

Function 19h

Get default drive

Function 25h

Set interrupt vector

Function 29h

Parse filename

Function 3000h

Get DOS version

Function 3306h

Get true DOS version

Function 34h

Get InDOS flag address

Function 35h

Get interrupt vector

Function 3700h

Get switch character

Function 3Ch

Create file

Function 3Dh

Open file

Function 3Eh

Close file

Function 3Fh

Read from file

Function 40h

Write to file (Used to write to stdout too)

Function 41h

Delete file

Function 42h

Seek in file

Function 45h

Duplicate file handle

Function 4400h

Used in initialisation to determine whether handle is to a device

Function 440Dh

Used to lock and unlock drives by L or W commands

Function 48h

Allocate memory

Function 4Ah

Resize memory

Function 4B01h

Load executable and return to debugger

Function 4Ch

Terminate process

Function 4Dh

Get process return code

Function 50h

Set PSP

Function 51h

Get PSP

Function 52h

Get List of Lists

Function 55h

Create child PSP

Function 58h

Get or set memory allocation strategy and UMB link status

Function 5D06h

Get DOS SDA address (used to switch active PSP)

Function 6Ch

Extended open/create

Function 716Ch

Extended open/create with LFN

Function 71A0h

Get LFN volume information

Function 7305h

Read/write sectors from/to DOS drive. Used to implement L and W command.

12.18 Interrupt 67h

EMS services. Used by X commands.

Section 13: Command help

13.1 lDebug help

```
lDebug (YYYY-MM-DD), debugger.

Usage: LDEBUG[.COM] [/C=commands] [[drive:][path]progname.ext [parameters

/C=commands   semicolon-separated list of commands (quote spaces)
/B            run a breakpoint within initialisation
/V[+|-]       enable/disable video screen swapping
progname.ext  (executable) file to debug or examine
parameters    parameters given to program

For a list of debugging commands, run LDEBUG and type ? at the prompt.
```

13.2 INSTSECT help

```
INSTSECT: Install boot sectors. 2018 by 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.

Options:

a:              load or update boot sectors of specified drive
/M=filename     operate on FS image file instead of drive
/MN             operate on drive instead of image file (default)
/MS=number      set sector size of FS image file (default 512)
/MO=number      set offset in image file in bytes (default 0)
/MOx=number     set offset (x = S sectors, K 1024, M 1024 * 1024)

/Fx=filename    replace Xth name in the boot sector, X = 1 to 4
/F=filename     alias to /F1=filename

/U KEEP         keep default/current boot unit handling (default)
/U AUTO         patch boot loader to use auto boot unit handling
/U xx           patch boot loader to use XXh as a fixed unit

/SR             do not read boot sector from source file (default)
/S=filename     read boot sector loader from source file
```

```
/S12=filename   as /S=filename but only for FAT12 (also /S16, /S32)
/SV             validate boot sector jump and FS ID (default)
/SN             do not validate boot sector jump and FS ID

/BS             write boot sector to drive's boot sector (default)
/B=filename     write boot sector to file, not to drive
/BN             do not write boot sector
/BR             replace boot sector loader with built-in one (default)
/BO             keep original boot sector
/BC             restore boot sector from backup copy

Only applicable for FAT32 with sector size below or equal to 512 bytes:

/IS             write FSIBOOT to drive's FSINFO sector (default)
/I=filename     write FSIBOOT to file, not to drive
/IB             write FSIBOOT to boot sector file (see /B=filename)
/IN             do not write FSIBOOT
/IR             replace reserved field with built-in FSIBOOT (default)
/IO             keep original reserved fields (including FSIBOOT area)
/IC             restore FSINFO from backup copy
/IZ             zero out reserved fields (including FSIBOOT area)
/II             leave invalid FSINFO structure
/IV             make valid FSINFO if there is none (default)

Only applicable for FAT32:

/C              force writing to backup copies
/CB             force writing sector to backup copy
/CI             force writing info to backup copy
/CN             disable writing to backup copies
/CNB            disable writing sector to backup copy
/CNI            disable writing info to backup copy
/CS             only write backup copies if writing sectors (default)
/CSB            only write sector to backup copy if writing sector
/CSI            only write info to backup copy if writing sector
```

Section 14: Online help pages

14.1 ? - Main online help

```
lDebug (YYYY-MM-DD) 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
 swap           BS index1 index2
 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[R][M][L] 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 [...]]
base display    H BASE=number [GROUP=number] [WIDTH=number] value
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
```

```
quit process    QA
quit and break  QB
register        R [register [value]]
Run R extended  RE
RE commands     RE.LIST|APPEND|REPLACE [commands]
Run Commandline RC
RC commands     RC.LIST|APPEND|REPLACE [commands]
MMX register    RM
FPU register    RN
toggle 386 regs RX
search          S range [REVERSE] [RANGE range|list]
sleep           SLEEP count [SECONDS|TICKS]
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]
view screen     V [ON|OFF [KEEP|NOKEEP]]
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
```

14.2 ?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
```

Enter ?F to display the recognized flags.

14.3 ?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.

14.4 ?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.

14.5 ?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.

14.6 ?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.

14.7 ?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.

14.8 ?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.

14.9 ?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
- 010000 R: highlight changed digits (needs ANSI for DOS output)
- 020000 R: highlight escape sequences to int 10h, else video attributes
- 040000 R: highlight changed registers (overrides 010000)
- 080000 R: include highlighting of EIP
- 100000 set PM ss B bit
- 200000 break on entering Protected Mode
- 02000000 do not call int 2F.1680 for idling
- 04000000 delay for a tick before writing breakpoints
- 08000000 do not call other lDebug instance's Update IISP Header call
- 10000000 disable auto-repeat
- 20000000 check int 16h buffer for Control-C if inputting from int 16h
- 40000000 call DOS service 0Bh to check for Control-C
- 80000000 when Q command is used while TSR, leave TF as is

More options: (read/write DCO4, read DCS4)

- 0002 enable interrupt 2Fh hook while in 86 Mode
- 0004 enable interrupt 8 hook
- 0008 enable interrupt 2Dh hook
- 00010000 force serial interrupt unhooking
- 00020000 force interrupt 2Fh unhooking
- 00040000 force interrupt 8 unhooking
- 00080000 force interrupt 2Dh unhooking
- 01000000 force interrupt 0 unhooking
- 02000000 force interrupt 1 unhooking
- 04000000 force interrupt 3 unhooking
- 08000000 force interrupt 6 unhooking
- 10000000 force interrupt 18h unhooking
- 20000000 force interrupt 19h unhooking

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
- 80 Disassembler: NEC V20 repeat rules (for segregs)

14.10 ?BOOT - Boot loading

Boot loading commands:

- BOOT LIST HDA
- BOOT DIR [partition] [dirname]
- 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

14.11 ?BUILD - lDebug build (only revisions)

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

14.12 ?B - lDebug build (with options)

```
lDebug (YYYY-MM-DD)
Source Control Revision ID: hg xxxxxxxxxxxx (vvvv ancestors)
Uses yyyyyyyy: Revision ID hg zzzzzzzzzzzz (www ancestors)
[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
```

14.13 ?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
```

14.14 ?SOURCE - lDebug source reference

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

Releases of lDebug are available via the website at https://pushbx.org/ecm/web/#projects-ldebug

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

14.15 ?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 15: 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.

15.1 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.

15.2 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.

15.3 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.

15.4 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.

15.5 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.

15.6 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.

15.7 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.

15.8 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.

15.9 LZSA2 depacker usage conditions

8086 Assembly lDOS iniload payload LZSA2 depacker

by C. Masloch, 2021

based on:

decompress_small.S - space-efficient decompressor implementation for 8088

Copyright (C) 2019 Emmanuel Marty

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.

15.10 aPLib depacker usage conditions

8086 Assembly lDOS iniload payload aPLib depacker

by C. Masloch, 2021

based on:

aplib_8088_small.S - size-optimized aPLib decompressor for 8088 - 145 bytes

Copyright (C) 2019 Emmanuel Marty

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.

15.11 bzpack depacker usage conditions

8086 Assembly lDOS iniload payload bzpack depacker

by C. Masloch, 2021

BSD 2-Clause License

Copyright (c) 2021, Milos Bazelides

All rights reserved.

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 SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Source Control Revision ID

hg e0af210a24c6, from commit on at 2022-09-24 13:15:01 +0200

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/ldebug/rev/e0af210a24c6