

IDebug manual

2020--2024 by E. C. Masloch. Usage of the works is permitted provided that this instrument is retained with the works, so that any entity that uses the works is notified of this instrument. **DISCLAIMER: THE WORKS ARE WITHOUT WARRANTY.**

This document has been compiled on 2024-09-15.

Contents

Section 1: Overview and highlights	20
1.1 Quick start for reading this manual	20
1.2 Some tips for using the debugger	21
Section 2: News	24
2.1 Release 9 (future)	24
2.2 Release 8 (2024-03-08)	25
2.3 Release 7 (2024-02-16)	26
2.4 Release 6 (2023-08-26)	27
2.5 Release 5 (2023-03-08)	30
2.6 Release 4 (2022-03-08)	36
2.7 Release 3 (2021-08-15)	37
2.8 Release 2 (2021-05-05)	39
2.9 Release 1 (2021-02-15) and earlier	40
Section 3: Building the debugger	42
3.1 Components for building	42
3.2 How to build	44
3.2.1 How to build the mktables program and the debugger tables	46
3.2.2 How to build the instsect application	48
3.2.3 How to prepare the test suite	48
3.3 Build options	49
Section 4: Getting started with the release	53
Section 5: Invoking the debugger	56
5.1 Invoking the debugger in boot loaded mode	56
5.2 Invoking the debugger as an application	56
5.3 Invoking the debugger as a device driver	59

5.4 Invoking the test suite	60
Section 6: Interface Reference	62
6.1 Interface Output	62
6.2 Interface Input	62
6.3 Enabling serial I/O	62
6.4 Register dumping	63
6.5 Memory dumping	64
6.6 Disassembly	64
6.7 Loading the debuggee	65
6.8 Running the debuggee	65
6.9 Help	66
Section 7: Debugging the debugger itself	67
7.1 Initialising the debuggable debugger	67
7.2 Sectioning overview	68
7.3 Debugging ELDs	70
7.3.1 Using the offset hint to debug ELDs	71
7.3.1.1 Using the hint ELDs	71
7.3.2 Using houdinis to debug ELDs	71
7.3.3 ELD examples	71
Section 8: Parameter Reference	74
8.1 Number	74
8.2 Address	74
8.3 Range	74
8.4 Range with LINES keyword allowed	75
8.5 List	75
8.6 List or range	76
8.7 Keyword	76
8.8 Index	76
8.9 Segment	76
8.10 Breakpoint	76

8.11 Label	76
8.12 Port	76
8.13 Drive	76
8.14 Sector	77
8.15 Condition	77
8.16 Register	77
8.17 Command	77
8.18 ID	77
8.19 Filename or pathname	77
8.20 Command line tail	77
Section 9: Expression Reference	79
9.1 Literals	79
9.2 String literals	79
9.3 Variables	79
9.4 Indirection	79
9.5 Parentheses	79
9.6 LINEAR keyword	79
9.7 DESCType keyword	80
9.8 VALUE IN construct	80
9.8.1 VALUE IN construct keywords	80
9.9 Conditional ?? :: construct	81
9.10 Expression side effects	81
Section 10: Command Reference	82
10.1 Empty command - Autorepeat	82
10.2 ? command	83
10.3 : prefix - GOTO label	84
10.4 . (dot) command - Immediate assembler	84
10.5 A command - Assemble	84
10.6 ATTACH command - Attach to process (Leave TSR mode)	84
10.7 B commands - Permanent breakpoints	85

10.7.1 BP command - Set breakpoint	86
10.7.2 BI command - Set breakpoint ID	87
10.7.3 BW command - Set breakpoint condition	87
10.7.4 BO command - Set breakpoint preferred offset	87
10.7.5 BN command - Set breakpoint number	87
10.7.6 BC command - Clear breakpoint	87
10.7.7 BD command - Disable breakpoint	88
10.7.8 BE command - Enable breakpoint	88
10.7.9 BT command - Toggle breakpoint	88
10.7.10 BS command - Swap breakpoint	88
10.7.11 BL command - List breakpoints	88
10.8 BU command - Break Upwards	89
10.9 BOOT commands - Boot loading support	90
10.9.1 BOOT PROTOCOL= command	90
10.9.1.1 Specify protocol	90
10.9.1.2 Altering protocol parameters	90
10.9.1.3 Specifying protocol partition	92
10.9.1.4 Specifying protocol filenames	94
10.9.1.5 Specifying protocol command line	94
10.9.1.6 Boot load protocol compatibilities	94
10.9.1.6.1 FreeDOS	94
10.9.1.6.2 DR-DOS	95
10.9.1.6.3 IBMDOS and MS-DOS 6	95
10.9.1.6.4 MS-DOS 7	95
10.9.1.7 Boot load protocol compatibilities additions	96
10.9.1.7.1 FreeDOS	96
10.9.1.7.2 DR-DOS	96
10.9.1.7.3 IBMDOS and MS-DOS 6	96
10.9.2 BOOT LIST command	96
10.9.3 BOOT DIR command	96

10.9.4 BOOT READ and BOOT WRITE commands	96
10.9.5 BOOT QUIT command	97
10.10 C command - Compare memory	97
10.11 COUNT command - Count list length	97
10.12 D command - Dump memory	97
10.13 DI command - Dump Interrupts	98
10.14 DM command - Dump MCBs	99
10.15 DZ/D\$/D#/DW# commands - Dump strings	100
10.16 D.A/D.D/D.B/D.L/D.T commands - Descriptor modification	100
10.16.1 D.A command - Allocate descriptor	101
10.16.2 D.D command - Deallocate descriptor	101
10.16.3 D.B command - Set descriptor base	101
10.16.4 D.L command - Set descriptor limit	101
10.16.5 D.T command - Set descriptor type	101
10.17 DT command - Dump text table	101
10.18 E command - Enter memory	102
10.19 EXT command - Load and run an Extension for lDebug	103
10.19.1 Current ELDs	103
10.20 F command - Fill memory	108
10.21 G command - Go	109
10.22 GOTO command - Control flow branch	110
10.23 H command - Hexadecimal add/subtract values	110
10.24 I command - Input from port	111
10.25 IF command - Control flow conditional	111
10.26 INSTALL command - Install optional features	112
10.27 L command - Load Program	115
10.28 L command - Load Sectors	115
10.29 M command - Move memory	116
10.30 M command - Set Machine mode	116
10.31 N command - Set program Name	116

10.32 O command - Output to port	117
10.33 P command - Proceed	117
10.34 Q command - Quit	117
10.35 QA command - Quit attached process	118
10.36 QB command - Quit and break	118
10.37 R command - Display and set Register values	118
10.37.1 RE command - Run register dump Extended	119
10.37.2 RE buffer commands	119
10.37.3 RC command - Run Command line buffer	120
10.37.4 RC buffer commands	120
10.38 RH command - Display Register dump History steps	120
10.39 RM command - Display MMX Registers	121
10.40 RN command - Display FPU Registers	121
10.41 RX command - Toggle 386 Register Extensions display	122
10.42 RV command - Show sundry variables	122
10.43 RVV command - Show nonzero user-defined variables	122
10.44 RVM command - Show debugger segments	122
10.45 RVP command - Show process information	122
10.46 RVD command - Show device information	123
10.47 S command - Search memory	124
10.48 SLEEP command	124
10.49 T command - Trace	124
10.49.1 TP command - Trace/Proceed past string ops	125
10.50 TM command - Show or set Trace Mode	125
10.51 TSR command - Enter TSR mode (Detach from process)	125
10.52 U command - Disassemble	125
10.53 UNINSTALL command - Uninstall optional features	126
10.54 V command - Video screen swapping	126
10.55 W command - Write Program	126
10.56 W command - Write Sectors	126

10.57 X commands - Expanded Memory (EMS) commands	126
10.58 Y command - Run script file	127
10.58.1 Y command pathnames	127
10.58.1.1 Y command configuration pathes	127
10.58.1.2 Y command default scripts path	128
10.58.2 Y command labels	128
10.58.3 Y command InDOS interaction	128
10.59 Z commands - Symbolic debugging support	128
10.59.1 Z /S=size - Allocate, resize, or free symbol tables	128
10.59.2 Z STAT - Show symbol table statistics	129
10.59.3 Z ADD - Add a symbol	129
10.59.4 Z DEL - Delete a symbol	129
10.59.5 Z COMMIT - Commit temporary symbols	129
10.59.6 Z ABORT - Discard temporary symbols	129
10.59.7 Z LIST - List symbols	130
10.59.8 Z MATCH - Match symbols	130
10.59.9 Z RELOC - Relocate symbols	130
Section 11: Assembler Reference	131
11.1 Assembler comparison to MSDebug	131
11.2 Assembler comparison to NASM	131
11.3 Assembler roundtrip	132
11.4 Disassembly fields	132
11.5 Assembly fields	133
11.6 Assembly instruction reference	133
11.6.1 Assembler instruction mnemonics	133
11.6.2 Assembler operand types	133
11.6.3 A quick overview of most 8086 instructions	134
11.6.4 ALU 2-operand instructions	135
11.6.5 ALU 1-operand instructions	135
11.6.6 Multiplication and division	135

11.6.7 Shifts and rotates	136
11.6.8 Data movement	136
11.6.9 Stack	136
11.6.10 Branch	137
11.6.11 Flags	137
11.6.12 String	138
11.6.13 Port I/O	138
11.6.14 Special: Addresses and segments	139
11.6.15 Special: Prefixes	139
11.6.16 Special: BCD	139
11.6.17 Special	140
Section 12: Variable Reference	141
12.1 Registers	141
12.2 MMX registers - MMxy	141
12.3 Options	142
12.3.1 DCO - Debugger Common Options	142
12.3.2 DCS - Debugger Common Startup options	142
12.3.3 DIF - Debugger Internal Flags	142
12.3.4 DAO - Debugger Assembly Options	142
12.3.5 DAS - Debugger Assembly Startup options	142
12.3.6 DPI - Debugger Parent Interrupt 22h	142
12.3.7 DPR - Debugger PProcess	142
12.3.8 DPP - Debugger Parent Process	142
12.3.9 DPS - Debugger Process Selector	142
12.3.10 DPSPSEL - Debugger PSP Segment/Selector	142
12.4 Default step counts	143
12.5 Default lengths	143
12.6 Limits	143
12.6.1 RELIMIT - RE buffer execution command limit	143
12.6.2 RECOUNT - RE buffer execution command count	143

12.6.3 RCLIMIT - RC buffer execution command limit	144
12.6.4 RCCOUNT - RC buffer execution command count	144
12.7 Return Codes	144
12.7.1 RC - Return Code	144
12.7.2 ERC - Error Return Code	144
12.8 Addresses	144
12.8.1 A address (AAS:AAO)	144
12.8.2 D address (ADS:ADO)	144
12.8.3 Address behind R disassembly (ABS:ABO)	144
12.8.4 U address (AUS:AUO)	144
12.8.5 E address (AES:AEO)	144
12.8.6 DZ address (AZS:AZO)	144
12.8.7 D\$ address (ACS:ACO)	144
12.8.8 D# address (APS:APO)	144
12.8.9 DW# address (AWS:AWO)	145
12.8.10 DX address (AXO)	145
12.9 I/O configuration	145
12.9.1 IOR - I/O Rows	145
12.9.2 IOC - I/O Columns	145
12.9.3 IOCLINE - I/O Columns for splitting lines in <code>getinput</code>	145
12.9.4 IOS - I/O Circular Keypress Buffer Start	145
12.9.5 IOE - I/O Circular Keypress Buffer End	146
12.9.6 IOL - I/O Amount of Script Levels to Cancel	146
12.9.7 IOF - I/O Flags	146
12.10 I/O reading variables	146
12.11 Serial configuration	146
12.11.1 DSR - Debugger Serial Rows	146
12.11.2 DSC - Debugger Serial Columns	147
12.11.3 DST - Debugger Serial Timeout	147
12.11.4 DSF - Debugger Serial FIFO size	147

12.11.5 DSPVI - Debugger Serial Port Variable Interrupt number	147
12.11.6 DSPVM - Debugger Serial Port Variable IRQ Mask	147
12.11.7 DSPVP - Debugger Serial Port Variable base Port	147
12.11.8 DSPVD - Debugger Serial Port Variable Divisor latch	147
12.11.9 DSPVS - Debugger Serial Port Variable Settings	147
12.11.10 DSPVF - Debugger Serial Port Variable FIFO select	147
12.12 Timer configuration	148
12.12.1 GREPIDLE - getc repeat idle count	148
12.12.2 SREPIDLE - Sleep repeat idle count	148
12.12.3 SMAXDELTA - Maximum encountered delta ticks	148
12.12.4 SDELTALIMIT - Delta ticks limit	148
12.13 _DEBUG1 variables	149
12.13.1 TRx - Test Readmem variables	149
12.13.2 TWx - Test Writemem variables	149
12.13.3 TLx - Test getLinear variables	149
12.13.4 TSx - Test getSegmented variables	150
12.14 _DEBUG3 variables	150
12.14.1 MT0 - Mask Test 0	150
12.14.2 MT1 - Mask Test 1	150
12.15 Y command variables	150
12.15.1 YSF - Y Script Flags	150
12.16 V variables - Variables with user-defined purpose	150
12.17 PSP variables	151
12.17.1 PSP - Process Segment Prefix	151
12.17.2 PPR - Process PaRent	151
12.17.3 PPI - Process Parent Interrupt 22h	151
12.17.4 PSPSEL - PSP segment or selector	151
12.18 SR variables - Search Results	151
12.18.1 SRC - Search Result Count	151
12.18.2 SRS - Search Result Segment	151

12.18.3 SRO - Search Result Offset	151
12.19 Access variables	151
12.19.1 READADR	151
12.19.2 READLEN	152
12.19.3 WRITADR	152
12.19.4 WRITLEN	152
12.20 Machine type variables	152
12.21 LFSR variables	152
12.22 RIxy - Real 86 Mode Interrupt vectors	153
12.23 FL.xF - Flag status	153
12.24 HHRESULT - H command result	154
12.25 DARESLT - D.A command result	154
12.26 XARESLT - XA command result	154
12.27 INT8CTRL - Interrupt 8 Control pressed detection time	154
12.28 Device mode variables	155
12.29 QQCODE - Q command termination return code	155
12.30 TERMCODE - Debuggee termination return code	155
12.31 DDTEXTAND - Data dump text AND mask	155
12.32 AMIS variables	155
12.32.1 TRYAMISNUM	155
12.32.2 AMISNUM	155
12.32.3 TRYDEBUGNUM	155
12.32.4 DEBUGFUNC	156
12.33 COUNT - List length count	156
12.34 RHCOUNT - Count of RH buffer entries	156
12.35 ELDAMOUNT - Amount of installed ELDs	156
12.36 CIP - Current CS's EIP or IP	156
12.37 CSP - Current SS's ESP or SP	156
12.38 Boot loading variables	157
12.38.1 BOOTUNITFLxx	157

Section 13: Interrupt Reference	158
13.1 Mandatory interrupt hooks	158
13.2 Serial interrupt	158
13.3 Interrupt 2Fh - Multiplex (DPMI entrypoint)	159
13.4 Interrupt 8 - Timer	159
13.5 Interrupt 2Dh - Alternate Multiplex Interrupt	159
13.5.1 AMIS private function 30h - Update IISP Header	160
13.5.2 AMIS private function 31h - Install DPMI entrypoint hook	161
13.5.3 AMIS private function 32h - Reserved for IDebugX	161
13.5.4 AMIS private function 33h - Install fault areas	161
13.5.5 AMIS private function 40h - Display message	162
13.5.6 AMIS private function 41h - Query message status	162
13.5.7 AMIS private function 42h - Get other link data	162
13.5.8 AMIS private function 43h - Inject a debugger command	163
Section 14: Service Reference	164
14.1 Interrupt 10h	164
14.2 Interrupt 16h	164
14.3 Interrupt 2Fh	164
14.4 Interrupt 12h	165
14.5 Protected Mode Interrupt 31h	165
14.6 Protected Mode Interrupt 2Fh	167
14.7 Protected Mode Interrupt 21h	167
14.8 Protected Mode Interrupt 25h	167
14.9 Protected Mode Interrupt 26h	167
14.10 Interrupt E6h	167
14.11 Interrupt 15h	167
14.12 Interrupt 13h	168
14.13 Interrupt 19h	168
14.14 Interrupt 2Dh	168
14.15 Interrupt 25h	168

14.16 Interrupt 26h	169
14.17 Interrupt 21h	169
14.18 Interrupt 67h	172
Section 15: Extensions for lDebug reference	173
15.1 LDMEM - Dump lDebug memory use.	173
15.2 HISTORY - Command history utility.	175
15.3 DI - Dump Interrupt vectors.	175
15.4 DM - Dump MCBs.	175
15.5 RN - Display FPU registers.	175
15.6 RM - Display MMX registers.	176
15.7 X - EMS commands.	176
15.8 DX - Dump Extended memory.	176
15.9 INSTNOUN - Operate on INSTALL flag nouns.	176
15.10 RECLAIM - Reclaim unused ELD memory.	177
15.11 ELDCOMP - Compare ELDs with differing linker options.	177
15.12 AFORMAT - Format assembly output.	178
15.13 AMISMSG - Display message received on AMIS interface.	178
15.14 AMOUNT - Provide ELDAMOUNT variable.	178
15.15 BASES - Convert between different numeric bases.	178
15.16 CO - Copy debugger terminal output to a file.	178
15.17 CONFIG - Access debugger config paths.	179
15.18 DTADISP - Displays the current DOS Disk Transfer Address.	180
15.19 IFEXT - Conditionally run a command if an ELD is installed.	180
15.20 KDISPLAY - Displays the current K/N command buffers' content.	180
15.21 LIST - List ELD/SLD files, description lines, sizes, help.	180
15.22 PRINTF - Print formatted output.	181
15.23 SET - Access environment variables.	181
15.24 USEPARAT - Display disassembly separators.	182
15.25 VARIABLE - Expand environment variables.	182
15.26 WITHHDR - Run commands with temporary DCO flags set.	182

15.27 AMISCMD - Run commands received on AMIS interface.	182
15.28 AMISOTH - Provide other link info on AMIS interface.	183
15.29 AMITSRS - List currently installed AMIS multiplexers.	183
15.30 BOOTDIR - List directory entries.	183
15.31 DBITMAP - Dump 8-bit-wide graphics from memory.	184
15.32 DOSCD - Change DOS current directory or drive.	184
15.33 DOSDIR - List directory entries.	185
15.34 DOSDRIVE - Get or set a DOS drive.	186
15.35 DOSPWD - Display DOS current directory.	186
15.36 EXTNAME - Guess EXT and Y command filename extensions.	186
15.37 INJECT - Inject commands into other debugger instance.	187
15.38 INSTNOTH - INSTNOUN which operates on other link debugger.	187
15.39 LDMEMOTH - LDMEM which operates on other link debugger.	187
15.40 LINFO - Display status of L command.	187
15.41 PATH - Path search for K/N commands.	188
15.42 EXTLIB - Library of ELDs.	189
15.43 EXTPAK - Compressed library of ELDs.	189
15.44 QUIT - Quit the machine.	189
15.45 DOSSEEK - Get or set the DOS 32-bit seek of a process handle.	190
15.46 ALIAS - Define aliases.	190
15.47 DPB - Display a DOS drive's DPB.	190
15.48 RDumpIdx - Dump text bytes pointed to by DS:SI and ES:DI in R register dump.	192
15.49 RDumpStr - Dump text pointed to by DS:DX in R register dump.	192
15.50 CHECKSUM - Calculate checksum over a memory range.	192
15.51 HINT - Display TracList hints to outer debugger	193
15.52 HINTOTH - Display TracList hints of the other link debugger	193
15.53 CHSTOOL - Work with int 13h partitions and geometry.	193
15.54 S - Search command with additional support for WILD and CAPS keywords	193
15.54.1 WILD - Search wildcard	193

15.54.2 CAPS - Search with capitalisation folding	194
15.54.3 UNCAPS - Reset search to do no capitalisation folding	194
15.54.4 S ELD internals	194
15.54.4.1 S ELD - Byte scan functions	194
15.54.4.2 S ELD - Trailing string comparison function	194
Section 16: Extension for lDebug format	196
16.1 ELD executable format	196
16.2 ELD instance format	197
16.3 ELD link info format	198
16.4 ELD link call table	199
16.5 ELD linker internals	200
16.5.1 ELD data macros	201
16.5.2 ELD code macros	201
16.5.3 ELD linker sources	202
16.5.4 ELD two-pass linker	203
16.6 ELD interfaces	203
16.6.1 ELD code and data buffers	203
16.6.2 ELD command handler	204
16.6.2.1 Procedure for installing ELD command handler	204
16.6.2.2 Procedure for uninstalling ELD command handler	205
16.6.3 ELD command injection	205
16.6.4 ELD preprocess handler	206
16.6.5 ELD AMIS handler	207
16.6.6 ELD multi-purpose puts handler	207
16.6.6.1 ELD multi-purpose puts handler: puts_ext_next entrypoint	208
16.6.7 ELD puts copyoutput handler	208
16.6.8 ELD puts getline handler	209
16.6.9 ELD variables	210
16.6.10 ELD near transfer interface	211

Section 17: Command help	212
17.1 IDebug help	212
17.2 INSTSECT help	212
Section 18: Online help pages	215
18.1 ? - Main online help	215
18.2 ?R - Registers	216
18.3 ?F - Flags	217
18.4 ?C - Conditionals	217
18.5 ?E - Expressions	218
18.6 ?V - Variables	219
18.7 ?RE - R Extended	219
18.8 ?RUN - Run keywords	220
18.9 ?OPTIONS - Options pages	220
18.10 ?O - Options	220
18.11 ?BOOT - Boot loading	225
18.12 ?BUILD - IDebug build (only revisions)	227
18.13 ?B - IDebug build (with options)	227
18.14 ?X - EMS commands	228
18.15 ?SOURCE - IDebug source reference	228
18.16 ?L - IDebug license	228
Section 19: Comparison of IDebug to MS-DOS Debug	230
Section 20: Test Reference	233
20.1 test_beep	233
20.2 test_build	233
20.3 test_rh	233
20.4 test_dt	233
20.5 test_rr_status	233
20.6 test_aa_basic	233
20.7 test_rr_basic	233
20.8 test_misc	233

20.9 test_timeout	234
20.10 test_int2D_unhook	234
20.11 test_bb_gg	235
20.12 test_bb_fill	235
20.13 test_access_var	235
20.14 test_dpmimini	235
20.15 test_dpmioffs	235
20.16 test_dpmialoc	235
20.17 test_missing_executable	236
20.18 test_error_executable	236
20.19 test_load_boot	236
20.20 test_yy	236
20.21 test_double_ctrlc	237
20.22 test_eee_interactive	237
20.23 test_rc	237
20.24 test_ext_extlib	237
20.25 test_ext_ldmem	237
20.26 test_ext_aformat	237
20.27 test_ext_checksum	237
20.28 test_ext_list	238
20.29 test_ext_amitsrs	238
20.30 test_ext_reclaim	238
20.31 test_ext_amount	238
20.32 test_ext_alias	238
20.33 test_ext_dosseek	238
20.34 test_ext_history	238
20.35 test_ext_amismsg	238
20.36 test_ext_amiscmd	239
20.37 test_ext_amisoth	239
Section 21: Additional usage conditions	240

21.1 GLaBIOS font license (used for dbitmap.eld)	240
21.2 BriefLZ depacker usage conditions	240
21.3 LZ4 depacker usage conditions	241
21.4 Snappy depacker usage conditions	241
21.5 Exomizer depacker usage conditions	241
21.6 X compressor depacker usage conditions	241
21.7 Heatshrink depacker usage conditions	242
21.8 Lzd usage conditions	242
21.9 LZO depacker usage conditions	242
21.10 LZSA2 depacker usage conditions	243
21.11 aPLib depacker usage conditions	243
21.12 bzipack depacker usage conditions	244
Source Control Revision ID	245

Section 1: Overview and highlights

IDebug 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, auto-repetition of some commands, and a number of extensions. There is also a symbolic debugging option being developed.

1.1 Quick start for reading this manual

- The interface reference explains the basics of the debugger's interface and lists some common commands.
- The parameter reference lists the types of parameters used by the available commands.
- The command reference describes most commands in detail.
- The expression reference details how numeric parameters are parsed by the expression evaluator.
- The variable reference lists a subset of the debugger's variables and what they can be used for.
- The assembler reference describes the assembly language used by the debugger's assembler and disassembler some.
- The interrupt reference lists what interrupt hooks the debugger sets up.
- The service reference lists what services are called by the debugger, which may be useful for developers of the debugger, or of kernels, ROM-BIOSes, or DPMI hosts.
- The Extensions for IDebug reference describes ELDs in some detail.
- The online help pages provide some additional descriptions not found elsewhere in the manual, as well as overviews of many different topics.
- The command help lists most of the switches and parameters accepted by the debugger and the instsect program.
- The news lists an overview of changes since prior releases of IDebug.
- Invoking the debugger states how to start the debugger, either bootloaded, as a device driver, as an application, or through the test suite.
- Building the debugger lists components, build options, and instructions on how to build.
- Debugging the debugger itself lists some considerations for working with CDebug or

DDebug as a debuggee.

- The test reference describes the tests that are implemented in the test suite.
- The additional usage conditions section lists attribution and licenses for the various depackers that can be used for the compressed debugger executable.
- Source Control Revision ID gives the Mercurial hash of the manual's source revision, and links to that revision in ecm's repository.

For a tour, definitely start with the interface reference. To help invoke the debugger read the section on 'Invoking the debugger as an application'. It may help to read the manual while testing the debugger on another terminal. Use the main page of the online help as a reference to what is possible, then check the command reference for details. To explain what parameter types are used refer to the parameter reference. For how to calculate refer to the expression reference.

Reconfiguring the debugger can be done using variables, including the Debugger Assembler Options (DAO) and the Debugger Common Options (DCO) 1 to 7. Use the variable reference and the online help pages on the options for those. Change variables using the R commands, such as 'r dco or= 800' or 'r ior := #50'. Some configuration can be done using the INSTALL command, refer to section 10.26.

1.2 Some tips for using the debugger

- At a '[more]' prompt for pagination, entering Ctrl-C aborts the current command and returns to the debugger command line.
- Use the GT or GNT commands to skip past conditionals
- Use G ABO to skip past calls or loops if P would not work
- T, P, G, U, and D commands have autorepeat to repeat the same command if an empty line (only blanks) is entered after they return control to the debugger prompt
- The POINTER type expression allows using a 32-bit number as a 16:16 segmented address wherever an address parameter is parsed
- The assembler can access the expression evaluator by surrounding an expression with parentheses '(. . .)'
- The A, E, D, and U commands write to the AAO, AEO, ADO, and AUO variables to point past the end of their last read or write
- The ABO variable points past the last instruction disassembled by an R command
- Permanent breakpoints can be set up using the B commands, including as pass points or conditionally
- The G command can repeat the prior list by specifying the AGAIN keyword as the first breakpoint, and can save to the list without actually executing the debuggee by ending the breakpoint list with a REMEMBER keyword
- T, P, and G commands can change either the instruction pointer only or also the code segment using an equals-sign-prefixed address as the first parameter. The TTEST command will change the CS:IP without actually executing anything.

- The machine type can be viewed or changed with an M command. Assembly and disassembly will note if a required machine level is absent according to the machine type.
- DCO option 800 or INSTALL GETINPUT enables the line editor and input history even when using DOS for input
- DCO6 option 200 or INSTALL BIOSOUTPUT will use the ROM-BIOS for output instead of DOS, including for register change highlighting
- DCO option 8 or INSTALL INDOS will act as if the debugger is always running with the InDOS flag set, avoiding DOS calls from the debugger itself
- The INSTALL AMIS and INSTALL TIMER commands can be used to provide the debugger's AMIS interface and hook the timer interrupt
- The DW ss:csp command is useful to view the current stack formatted as words
- The debugger is largely capitalisation-insensitive
- The LFSR variable allows to generate a stream of pseudo-random numbers
- The DIM command shows MCB names of blocks pointed to by interrupt vectors, while DIL (or DIML) will query AMIS multiplexers for their interrupt entrypoints to find hidden chains
- Long output of many commands is paged by default, displaying a '[more]' prompt that pauses the output until a keypress is received
- The VALUE IN construct allows to match one numeric value or range against many match values or match ranges
- A breakpoint can be set up on an interrupt handler by issuing for example 'BP NEW ptr ri2Dp', if the interrupt handler is writeable
- If given a single expression the H command displays the result as hexadecimal and as decimal
- Decimal numbers can be entered with a # prefix, and binary with 2#. Character codes can be used as numbers with #" . . . "
- Script files can be run with the Y command
- A program can be traced until it tries to modify interrupts 1 or 3 using 'tp FFFFF while ! value from linear 0:1*4 length 3*4 in writing silent'. Add a conditional breakpoint like 'bp new ptr ri2lp when value ax in 2501, 2503' beforehand to intercept DOS calls to change these interrupts. (ICDebug or IDDebug require INSTALL INDOS to write a breakpoint on the int 21h handler.)
- If at its entrypoint and you want to return from a 16-bit near function, use 'g word [ss:sp]'. For a 16-bit far or interrupt function use 'g ptr [ss:sp]'
- In a loop with several exit conditions, trace with T or P and accumulate exits with 'G AGAIN address REMEMBER' (insert an offset or (NOT) TAKEN keywords for the address), then finally run 'G AGAIN'

- The L and W commands for reading and writing sectors accept drive letters for their second parameter, specified with a trailing colon
- The RV, RVM, RVP, and RVD commands will show some information on modes, memory segment locations, processes, and device headers
- RX toggles the 32-bit register view for the R command
- The RE buffer can contain many different commands, which are run on RE commands or when T, P, or G initiate a register dump
- The DCO options are described in the full ?O help page as well as individual pages like ?O6 for DCO6
- Not sure what version of IDebug is running? The command ?BUILD displays the description and source control revision IDs. The description includes a build date or release number.
- Setting a breakpoint at the current instruction pointer with the G command will trace past this instruction once then write the breakpoint and run at full speed
- There's a TSR mode converting the application-mode debugger into a resident program, allowing to debug the debugger's parent process
- The symbolic F register for the R command allows dumping and modifying the flags using the abbreviated flag states. Use the ?F command to list the meanings of the flag states.
- The R command can modify variables using binary operators suffixed by an equal sign, both on the command line of the R command as well as after the R variable prompt
- By adding a trailing dot after a variable name for the R command, a variable's value can be inspected without bringing up a variable prompt that requires submitting a second input line
- A single dot input can exit any special prompt, such as the interactive enter mode, the assembler, or the R command variable prompt

Section 2: News

2.1 Release 9 (future)

- Disable DX command by default, as the code segment ran out of space in the ICDebugX build. The command can still be used as an Extension for IDebug.
- Add ELD1TAIL trailer header support, so that an ELD may be appended to another file and the ELD loader will find it by seeking from the EOF
- Add test reference to manual
- Fix, do not eat comma after length number if no length keyword follows
- Add boot load setting DRDOS (same as IBMDOS with `maxpara=-1`)
- Allow to boot kernel of up to 29 KiB to linear 00700h by shrinking the stack reservation below the BPB to 256 Bytes (or 512 Bytes if passing a command line)
- Work around Book8088 / Xi8088 BIOS bug when trying to detect LBA support in bootloaded debugger
- Fix, allow to run installed ELD commands in the THEN clause of IF commands
- Fix to not corrupt device driver's loader PSP
- Bugfix, correctly parse device driver command line ending in LF
- Add assembler reference chapter to manual
- In assembler disable optimisation from index with scale times 1 to base, now acting like NASM with `nosplit` keyword
- Display scale times 1 explicitly in disassembler
- Fix in assembler to always allow blanks after scale
- Add DAO flag 2_0000h to disassemble in NASM style using ECX or CX as second operand to `loop` instructions if ASIZE prefix is present, rather than D or W suffix
- Add DAO flag 1_0000h to hide MODRM keywords
- Match NASM and NDISASM order of operands for `xchg` with a ModR/M, in both the assembler and disassembler
- Assemble and disassemble MODRM keyword to select or indicate non-default forms of instructions
- Reject `movzx` to 32-bit register with memory source lacking a size keyword

- Allow to assemble displacement size keyword within brackets
- In many cases disassemble immediates with a size keyword if they are encoded in a longer form than needed
- Allow to assemble `JMP FAR imm` with a single immediate, which becomes the offset that is combined with the AAS as a segment
- Allow to specify `BYTE 1` to assembler to use an imm8 shift/rotate count (the 186+ form) similar to NASM, and disassemble this with the `BYTE` keyword as well
- Optimise to sign-extended 8-bit immediate/displacement when an appropriate 16-bit value is specified to the assembler (eg `'adc dx, FFFF'`, `'imul dx, dx, FFFF'`, `'push FFFF'`, `'mov dx, [bx + FFFF]'`)
- In assembler allow `'push [100]'` or `'pop [100]'` without a size specified, defaulting to word size (in 86 Mode or a 16-bit CS), or to dword size (in a 32-bit CS for IDebugX). Disassembler however always shows the size.
- Fix, allow comma between number and length keyword
- IDebugX: Fix BL (breakpoint list) segment output in PM when not matching current CS
- Fix: Parsing a linear address starting with '@(' required two closing parens
- On too long lines of assembler directive `DD` (using numeric or string data) do not actually overflow the buffer, rather detect the overflow before the write
- IDebugX: Fix non-terminate PM to 86M switch with the selector in a seg/sel variable (eg ADS) not on a segment boundary
- IDebugX: Fix PM int 21h function 4Ch on 32-bit stacks

2.2 Release 8 (2024-03-08)

- Fix bootloaded `Y :label` command
- Add new tools `patchqry` and `patchpro` to patch default values in the iniload query patch site and in the incomp ICFG block to select a progress display choice (shipping in separate repo `patchini`, also provided as current build)
- Added progress displays to compressed executable depackers, set DOS environment variable `LDEBUGPROGRESS` to a number: 1 (dots), 2 (percentage), 3 (bar), 4 (bar and percentage)
- Exomizer on server updated to current git commit, incomp depacker updated to handle P & 32 flag
- Fix: Correct drive unlocked after `W` sector-writing command, and on MS-DOS v7.00 drive will be locked at all even though int 21h function 7305h is not used
- Fix: Passing a debuggee pathname to the device mode debugger, which is invalid, will no longer crash the machine and display an error instead
- Added a `rel.sh` script which will automate the release build. It configures the ELD mak

scripts to not create the XLDs nor eldcomp.eld, and deletes all listing files and many of the temporary files.

- Moved packlib output files into tmp directory rather than bin
- Fix: AMIS description is padded so as to keep most offsets identical between release builds and daily builds

2.3 Release 7 (2024-02-16)

- Fix bug when specifying BOOT PROTOCOL= with the second file (add file) in a subdirectory
- Allow to resize history buffer segment in init using the /H= switch
- ELD data blocks can be resized in init using the /Y= switch
- In path search (init or path.eld) skip directories
- Fix boot EXT/Y/BOOT DIR reads when FAT12 entry that straddles a sector boundary is read
- Support comma separator before program load name
- Add ::empty:: keyword to skip automatic scripts path search in EXT and Y commands
- Enable _EXTENSIONS by default
- Disable _EMS, _RN, and _RM by default
- Optimise application/device init memory use, requires a second init relocation for large buffers
- Allocate a separate environment block of size 2 KiB for the debugger
- Allow unquoted comma to separate Y and EXT filenames
- Support ::scripts:: and ::config:: prefixes in boot Y and EXT commands, as well as the BOOT DIR command
- Enable new NASM warnings to avoid section-crossing near and short branches
- Fix boot file read ending on both a cluster boundary and the End Of File
- Fix int 21h calls with DS != SS.
- Add Extension for lDebug loading and basic interface for linking. Still default disabled at build time. If enabled, an /X=MAX switch enlarges the ext segment buffer.
- Fix, truncate areas addresses for non-bootloaded mode.
- Fix a bug in /A switch relocation of code section (if layout 2 in use and _PM_DUALCODE enabled).
- Add /T switch to relocate debugger during init.
- Add a fractional digit to bytes size formatting.

- Fix: In DIL check for valid appearing int 2Dh before calling it.
- Fix: Do not allow DI with two parameters where the second is below the first.
- Add INSTALL TOGGLE command.
- Fix INSTALL command with AREAS keyword followed by another keyword.
- Fix: DIL command was broken since addition of the /A switch (enlarge auxiliary buffer) due to using wrong segment.
- A single list parameter can be specified with different sizes, switching back and forth using 'AS size' keywords.

2.4 Release 6 (2023-08-26)

- Add length keywords PAGES, KiB, MiB, GiB
- Call DOS_HELPER_PRESTROKES_START dosemu2 helper service
- Add byte length variables DENTRYLEN, DSTACKLEN, DMESSEGELEN, DCODE1LEN, DCODE2LEN, DAUXBUFLEN, DHISBUFLEN. Also DALLOCSEG and paragraph size DALLOCSIZE.
- Emit warning on unknown filename extension in INIT. Can be disabled with /PW- switch.
- /P switch to guess filename extension and do path search in INIT. /PS for path search only, /PE for guessing extension only.
- Make interrupt 0Dh, 0Ch hooks a run time option. Add INSTALL noun INTFAULTS to enable these hooks.
- Add CLEAR command.
- Serial I/O is controlled by an internal variable rather than directly by the DCO option. Fixes some bugs, such as running INSTALL SERIAL, TIMER, AMIS.
- In S command result data dump list displacements after the result address to indicate that the data dumped is *after* the match and does not include the match data itself.
- Add ::SCRIPTS:: path keyword for DOS I/O Y command. Also used if a Script for IDebug file is not found.
- Add H AS SIZE modes to display result using the decimal bytes size display (from DM command).
- Introduce /A= switch for larger auxiliary buffer in application mode or device mode.
- IOK variable added. Both IOI and IOK will force reading from a terminal now.
- IDebugX fix: In PM, if a segment is limited to 64 KiB, do not attempt to dump S command result data beyond the limit.
- Introduce the WHILE buffer, and use it also for G command. Also used for S command and RC./RE.REPLACE, unless it is in use in which case these commands can still use the auxiliary buffer (if it isn't in use).

- Add RH mode and RH command (none, one, two, or IN parameters). This mode uses the auxiliary buffer, disabling other commands that use it. Add RHCOUNT variable too.
- IDebugX fix: Do not corrupt buffered commands when running DX command.
- Fix, set error code if BOOT command attempts to read from 33-bit space using CHS addressing.
- Add COUNT command and variable. S command also sets the variable.
- Add TOP keyword for D, DB, DW, DD, and DX commands.
- Add DCO2 40_0000h for unconditional linebreak before R dump, changed DCO6 8000_0000h to be conditional on int 10h being used and the current column being nonzero.
- Add DCO2 20_0000h for underscores in 32-bit R variable display.
- Add ‘?VERSION’ help page.
- Add ‘END’ keyword to range parameter parsing.
- Add new INSTALL command keywords to aid reconfiguring the debugger.
- Application and device mode will now detect a startup Script for IDebug file, intended for configuration. The /IN switch disables this detection.
- Bugfix: Y command may have failed due to a line overflow.
- Improve detection of invalid DD commands that may be valid when interpreted as D commands.
- Display H command remainder if the last operation is a division.
- In `getinput` allow Ctrl-A (like Home) and Ctrl-E (like End).
- DT command (ASCII table if empty, else text of specified byte values). Additionally, DTT command (dump text table top half).
- IDebugX bugfix: If dumping for example 8 bytes at 1FFF0h in a large segment the debugger would loop infinitely.
- AS SIZE keywords for E, F, S strings.
- Added DAO option 400h for MSDebug style opcode field width in disassembly.
- Added DCO2 10_0000h to do some R command variable prompts in MSDebug style.
- Added DCO2 option 8_0000h to display additional blanks in R command register dump for MSDebug style.
- Added DCO2 option 40000h to treat explicit length 0 as 64 KiB, except U treats it as length 1, to improve MSDebug compatibility.
- Bugfix: In F RANGE command forbid large destination lengths if the source range is specified in a 16-bit (limit \leq 64 KiB) segment without a length specified. Would truncate the source to up to offset 0FFFFh.
- Quotemarks added as expression separators to improve MSDebug compatibility.

- Add IDebug ad section to manual (in section 19), comparing IDebug to MSDebug and original MS-DOS Debug.
- Allow to read .HEX files. Only record type 00h and 04h used. Can read files larger than 64 KiB, unlike MS Debug.
- Add K command (same as our default N command), and DCO2 options for MS Debug compatible N command.
- Add RC.ABORT command to allow a Script for IDebug file called from the RC command line buffer to modify the RC buffer itself.
- Add DCO6 options for style 2 and style 3 alternative flag state dump.
- Add DCO6 option to display a linebreak before R dump.
- New _EXPRDUALCODE build option to save some 5 kB in the first code segment. Disabled by default.
- Fix: Device mode debugger should not retain an open handle for FDCONFIG.SYS.
- Trying to open the LDEBUG\$\$ device will now cause a critical error.
- Some code in the run exit and entrypoints can be moved into entry segment, saving about 250 bytes in the first code segment.
- DPMI exception entrypoint handlers can be moved into entry segment, saving nearly 300 bytes in the first code segment.
- Fix: When a symbol table is allocated in DOS memory and the debugger quits with a QD command, the table would stay allocated.
- Add ATTACH command to attach to a PSP, can be used while application mode or device mode debugger is resident. (Opposite of TSR command.)
- MMX register support in expression evaluator optimised to take up less run time of other expression terms
- mktables program extended, allowing for builds of the debugger that do not contain table entries above specified machine level.
- Non-boot-loaded modes now discard the ?BOOT help page, saving 2.7 kB of resident memory use.
- Refactor to store long help messages in another segment, reducing the pressure of the data entry segment by 22 kB.
- Fix QC command if not last in container, would corrupt MCB chain
- Application and device driver mode now discard most of the boot loaded mode code, saving about 9 kB of resident memory use.
- Fix an IDebugX D command bug on non-386 machines
- Add build option _APPLICATION, to create special-purpose builds if disabled. (Not provided pre-built as yet.) \$RESULTTEXT variable for mak.sh can set a filename extension other than '.com'.

- getinput redraw optimised, fewer complete redraws
- Add DCO3 option 0100_0000h to highlight prefix/suffix in getinput if text parts are not visible
- Improve handling of very narrow displays
- RE.LIST and RC.LIST use the IOCLINE setting now

2.5 Release 5 (2023-03-08)

- Client PSP is now stored internally as a segment, even when lDebugX is in Protected Mode. DM command now shows this segment rather than a selector. (The PSPSEL variable can be read for a selector instead.)
- RM command now accepts an optional size keyword
- 'IF EXISTS R variable THEN' command added
- MMX support of the machine is re-detected after lDebugX switched modes
- Fix: RM command and MMxy variable read and R MMxy variable write work now
- Fix: In /E+ mode a zero word is now pushed to the initial stack
- dosemu2 -dumb mode now detected specifically when needed for register change highlighting to ROM-BIOS interrupt 10h
- Fix: Bootloaded init could have corrupted part of its image during second relocation, if needed. (This case cannot occur during normal load at 200h:0 on a machine with 512 KiB or more of available LMA.)
- Fix: Creating a process resets the DTA to PSP:80h
- Fix: 'R VD' accesses VD variable, rather than run 'RVD' command
- Improvements to support NEC V20 machine
- Add CIP and CSP variables, use CIP for EXECUTING keyword
- Fix error handling if S command search area is shorter than search mask
- IOCLINE variable added to support display on the HP 95LX's narrow screen
- Fix: Stack access variables set wrongly for interrupt instructions
- Fix: In DPMI protected mode QA or Q command sometimes would not work
- Immediate assembler merged. Currently defaults to disabled at build time.
- Bootable lDebug: Allow to access small FAT32 file system (less than 64 Ki clusters)
- Add timer configuration variables and default the SDELTA LIMIT variable to 5 to improve wait performance
- Serial I/O: Add option to always do EOI after calling downlink for IRQ sharing. Autodetect in KEEP prompt if this option must be used or if IRQ sharing must not be used.

- Query patch support for ldosboot iniload can be used to set the BOOTUNITFLx variable for the debugger load unit
- Change: DCO6 option 200h only makes output go to ROM-BIOS, new DCO6 option 0100_0000h makes all Input/Output use ROM-BIOS. (The 200h flag still allows to read script files from DOS.)
- lDebugX: Fix, allow to use TSR command in PM (expected a segment where a selector was read)
- lDebugX: Fix PSP variables and TSR command expecting higher limit in PM (getsegmented now sets limit 0)
- lDebugX: Fix getexpression not preserving the scratch selector
- Add FL.xF variables to read flag status in expressions, eg FL.CF which reads as 1 if CY and 0 if NC
- When creating an empty process (eg after QA command) also write the current command line tail to it (instead of the debugger's internal N buffers)
- Display amount of ancestors for the symsnip revision ID
- A pair of 32-bit E command fixes
- Write AES:AEO (e_addr) in E command and allow E command without an address
- lDebugX: Add DESCTYPE keyword in expressions
- lDebugX: Add descriptor modification commands and DARESLT variable, refer to section 10.16
- Add XARESLT variable for XA command
- Allow to share the serial IRQ so eg two lDebug instances can be connected to two serial ports that use the same IRQ, like COM2 and COM4 (both use IRQ #3 by default)
- Do not simulate repeated string scan/compare instructions when disassembling them using the U command (only do so for R command disassembly)
- Disassembler handles o32/o16 OSIZE prefixes as belonging to push and pop with segregs, instead of displaying the prefix as 'unused'.
- If simulation of repeated string scan/compare instructions is disabled in DAO then the access variables will now be set up assuming a count of 1, rather than the maximum possible count.
- Add convenience entrypoints for debuggable mode at CODE:1, CODE2:0, and CODE2:1. The offset 0 entrypoints will return to the main command loop, called 'cmd3'. The offset 1 entrypoints will additionally display a linebreak.
- Allow switching lCDebugX to debuggable mode upon putrunint and allow running a breakpoint early in putrunint.
- Add AMIS private function 33h, provided by lDebugX by default and can be used by lDDebugX/lCDebugX by default

- Add INSTALL and UNINSTALL commands
- Allow to specify length of D command with LINES keyword
- Add DEFAULTDLEN, DEFAULTDLINES, DEFAULTULEN, and DEFAULTULINES variables to modify default sizes of D and U commands
- Display long numeric constants with underscore separators for readability in online help pages and documentation
- Allow to disable disassembler memory access for referenced memory and repeated string instruction simulation, using four new DAO flags
- Change HP 95LX 40-column friendly mode support in the disassembler to use two DAO flags rather than two DCO6 flags (one of which was shared)
- Allow to specify variable RIXP/S/O/L with x as a single-digit hexadecimal number
- Bugfix: Allow operators between ?? and :: in a ternary operator expression for H command
- Allow to specify SILENT keyword followed by a number before the S command's range or list, display only up to a certain amount of results
- Add CLR operator, bitwise AND with the bitwise NOT of the right hand operand. Precedence above bitwise AND.
- Bugfix: Absolute value operator ? should always give its result an unsigned type
- Allow switching ICDebugX to debuggable mode upon debugger exception and allow running a breakpoint early or late in debugger exception.
- Add debugger exception areas to display the cause of a memory access which may fault in the debugger. Also adds a linebreak for eg referenced memory reads to work in tandem with the partial disassembler output. (Idea from FreeDOS Debug/X, though there it only does a linebreak and implements it differently.)
- Bugfix: Disassemble mov with segreg and memory operand always with m16, ignoring the operand size selected. In assembler never emit an osize prefix and reject an explicit dword size keyword.
- In disassembler display instruction and referenced memory address before accessing memory, so partial output is displayed in case a fault occurs in the debugger (FreeDOS Debug/X pick)
- Document side effects of expression evaluator
- Bugfix, allow all valid address parameter formats for the source address of an M move memory command (suggested by FreeDOS Debug/X)
- Add AMIS private function 31h to instruct IDebugX to try to install its DPMI hook, make use of this in IDDebugX and ICDebugX
- Add descriptions of BOOT commands to manual
- Fix HIDDEN= keyword for BOOT READ/WRITE with partition specified, add HIDDENADD= keyword to modify rather than replace hidden sectors

- Fix a bug in BOOT PROTOCOL= command that could disallow use of ENTRY and BPB parameters if running a non-DPMI build on a 386+ machine
- Avoid faults in the debugger if a code selector with a low limit gets used for writing, eg by A or E commands
- Fix bug setting wrong debuggee CS limit if _DUALCODE build
- Add variables DSTACKSEG/SEL, DENTRYSEG/SEL, DCODE1/2SEG/SEL, DAUXBUFSEG/SEL, DHISBUFSEG/SEL, DSCRATCHSEL, DSYM1/2SEL (selector variables only for IDebugX, symrels only for symbolic IDebugX)
- RVM command shows second code segment if _DUALCODE build
- In C, E, and A commands in PM display original selector, not the scratch selector as replacement. Variable AAS is also affected by this.
- Fix a bug with different selectors for IDebugX C and M commands (picked from FreeDOS Debug/X version 2.00)
- Add taken keywords to address parsing, refer to section 8.2. (This effectively adds the GT and GNT commands, as well as TTEST=TAKEN and TTEST=NOTTAKEN to change (e)ip.)
- Bugfix: Allow entering double-slash to disable second file search for the BOOT PROTOCOL= command even if no command line follows
- Application /C= switch and kernel command line will now skip leading blanks following a semicolon that is converted to a linebreak
- Modify serial interrupt handler to pass on interrupt call if the PIC does not indicate an interrupt in the In-Service Register (ISR)
- Add /M switch and interrupt 7, 0Ch, 0Dh hooks (for R86M exceptions), though build options for all of them are disabled by default
- Add force BPB CHS geometry (4) and force LBA access (2) flags to the BOOTUNITFLx flag variables
- Add switches /F and /E
- Allow zero as parenthetical partition specification, allow to access partition `u(bootldpunit).(bootldppart)` if LDP is equal to FDA
- Bugfix: Reading with BOOT command that crossed 64 KiB DMA boundary would copy too much or too little from the sector segment to target
- Start of HP 95LX support (NEC disassembly repeat rules, narrower R/U/D command output, do not intercept interrupt 6)
- Bugfix: Command `r f .` no longer displays garbage
- Use `test_high_limit` to check segment limits, to determine whether to use 32-bit offsets in several spots

- Disable calling XMS by Protected Mode far call by default
- Add dual code segment support to allow code size beyond 64 KiB
- Introduce dash prefix to commands to disable symbolic debugging features (no-op if not symbolic build)
- Introduce U address LENGTH length LINES keyword to disassemble a number of lines rather than bytes
- 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-redirectioned 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-redirectioned 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 DPML allocation while the client is running in 86 Mode)
- Fix some cases of detecting 32-bit offsets incorrectly

2.6 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 bzipack compression method
- Drop DPS variable when building without DPMM 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

2.7 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 IDebugX'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

2.8 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')

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

- 'G REMEMBER' command to work with the saved temporary breakpoint list
- WHEN conditions for permanent breakpoints
- RlxxO/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 1 10 0' \ 's 100 1 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 IDebug is not supported on conventional DOS-like systems. (DJGPP environments may suffice but are not tested.) Assembling the main debugger executable may require up to 1 GiB of memory.

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) and to filter reproduce dt file contents
- `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
- GNU time - when `$use_build_time` is enabled, to display main assembler run's memory use and wall time
- `python` - to run `hg` and to run the test suite
- C compiler - to compile supporting programs
- `dosemu2` - to run build decompression tests (optional)
- `qemu` - to run build decompression tests (optional)
- `nasm` - to assemble. NASM versions to choose:
 - NASM versions up to 2.07 fail -- `%def tok` is not supported
 - NASM versions prior to 2.09.02 fail -- `%def tok` is implemented wrongly
 - NASM version 2.09.02 works (last tested 2019-11)
 - NASM versions 2.09.03 to 2.09.10 all fail -- `%assign %$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)
- NASM version 2.16rc10 works (last tested 2022-11)
- Upcoming NASM version beyond 2.16.01 works (last tested 2023-07), with a patch to fix a major memory leak
- heatshrink - to build compressed extpak.eld (heatshrink can also be used to compress the incomp stage payload)
- halibut - to build this manual, with a patch to include the hash links next to section names
- supporting programs:
 - mktables (included in debugger source)
 - tellsize (included in separate repo called tellsize)
 - mktmpinc.pl (included in separate repo called mktmpinc, to create temporary include files, optional)
 - 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, checkpl if using new style checksumming)
 - instsect (application to install boot sector loaders)
 - booting (to run decompression test with qemu and create boot image for qemu to use for the test suite)
 - incomp (if to use compression support), also needs one of:
 - brieflz (blzpack)
 - lz4
 - 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` -- default choice, one of the fastest depackers
- `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 by setting it to a nonzero number. 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. If `use_build_decomp_test` is enabled, you may also enable `INICOMP_PROGRESS` (again by setting it to a nonzero number). This enables code for a progress display during depacking of a compressed executable. The default for application mode and device driver mode is to not display the progress indicator. Setting the (DOS) environment variable `LDEBUGPROGRESS` to a value between 1 and 4 will enable the indicator. This is now enabled by default in release and current daily builds.

9. Edit `INICOMP_WINNER` to name one method or the keyword 'smallest' or 'fastest'. The latter requires `use_build_decomp_test` to be enabled, as well as to set the variable `INICOMP_SPEED_TEST` to a nonzero number. This number specifies how often to decompress the image during the decompression timing test. A number of 16 or higher is recommended.
10. 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.
11. 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.
12. 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.
13. 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, EDR-DOS DRBIO.SYS, 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 or EDR-DOS load protocols impossible. All the other protocols work even in the presence of arbitrarily large appended data.

3.2.1 How to build the mktables program and the debugger tables

The mktables tool is a C program that prepares the debugger assembler/disassembler tables from three input files. The input files are:

instr.set

Instruction set, mapping mnemonics to opcodes and instruction keys

instr.key

Instruction keys, each one specifying a list of operand types

instr.ord

Instruction orderings, specifying which keys must be preferred by the assembler

The output files are:

debugtbl.inc

Receives the tables

debugtbl.old

Receives prior output file

debugtbl.tmp

Used during building the tables

As mentioned, mktables only needs to be built and ran if any of the input files, or mktables itself, or the mktables options have changed. The debugger's hg repo carries a copy of the current full default tables so running mktables is usually not needed.

The following options are available:

`[no]direction`

Enables or disables memory access direction operands. These operands tell the disassembler whether a memory operand is a source, a destination, or both. This option is on by default.

`[no]stackhinting`

Enables or disables stack access hint operands. These operands tell the disassembler that an instruction represents a stack push, stack pop, or stack special operation. This option is on by default.

`[no]merge`

Enables or disables merging redundant tail operand lists. This is a small optimisation of the operand list tables. This option is on by default.

`[no]debug`

Enables or disables debugging output. This option is off by default.

`filename`

Followed by a filename argument. The indicated file is used as output file instead of the default `'debugtbl.inc'`.

`[no]offset`

Enables or disables offset comments in output tables. Disabling them makes it easier to find changes between different tables without the noise added by them. This option is on by default.

`[no]discardunused`

Enables or disables discarding unused operand lists. Enabling helps optimise the output tables, particularly when a machine limit is in use. This option is on by default.

`8086, 186, 286, 386, 486, all`

Limit tables to include only instructions up to the specified machine type. Default is `'all'`.

mktables is built using a C compiler. OpenWatcom and gcc should both work. An example to build mktables on a DOS host with OpenWatcom is:

```
wcl -ox -3 -d__MSDOS__ mktables.c
```

An example to build mktables with gcc on a Linux host:

```
gcc -xc MKTABLES.C -o mktables -Wno-write-strings -DOMIT_VOLATILE_VOID
```

3.2.2 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/1st, 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 IDOS and the default kernel name LDEBUG.COM. Read the instsect help page for instructions on how to use it. Refer to section 17.2 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 IDOS 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).

3.2.3 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` or `intrtn1_entry.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_*`, 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. This DCO6 flag can also be changed using `INSTALL DEBUG` commands.

lDebug with the `_DEBUG` option disabled accepts a no-op `/D-` switch. lDDebug with `_DEBUG` enabled but `_DEBUG_COND` disabled accepts a no-op `/D+` switch.

`_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 '#'. (IDDebugX or ICDebugX in debuggable mode prepends its tilde to that resulting in '~#'.)

_APPLICATION

Enabled by default. Enables to use the executable as a DOS application. Disabling allows to save a little memory for a special-purpose build.

_DEVICE

Enabled by default. Enables to use the executable as a DOS device driver. Disabling allows to save a little memory for a special-purpose build.

_BOOTLDR

Enabled by default. Makes the program support being bootloaded. This additionally requires the IDOS 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 18.11. Disabling this option allows to save a little memory for a special-purpose build.

_HISTORY

Enables the line editing history for getinput (ROM-BIOS input or serial input, DOS input only if getinput enabled). 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. Size of a separate segment can now be modified by the init using the DOS device or application command line switch /H.

_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 READADDR0, READLEN0, etc. Refer to section 12.19. The access variables are written after an R command's register dump and disassembly (refer to section 10.37). Access variables can be accessed using special keywords behind the IN of a VALUE x IN y construct (refer to section 9.8).

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 10.50); 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. This may require the `_DUALCODE` build option to be enabled.

`_IMMASM`

Enable the immediate assembler. Refer to section 10.4. This currently defaults to off. This may require the `_DUALCODE` build option to be enabled.

`_DUALCODE`

Enable the second code segment of the debugger. Most of the symbolic and immediate assembler code will be put into the second code segment if it is enabled. Currently defaults to on if both `_PM` and `_SYMBOLIC` are enabled.

`_BOOTLDR_DISCARD`

Discard boot loaded mode specific code when installing the debugger as a DOS application or DOS device driver. This saves about 9 kB of resident memory use. Enabled by default.

`_BOOTLDR_DISCARD_HELP`

Discard boot loaded mode ?BOOT help page when installing the debugger as a DOS application or DOS device driver. This saves about 2.7 kB of resident memory use. Enabled by default. Only takes effect if option `_MESSAGESEGMENT` is enabled.

`_MESSAGESEGMENT`

Adds an additional segment for storing long messages like the help pages. This moves about 20 kB out of the data entry segment. Enabled by default.

`_EXPRDUALCODE`

Moves most code of the expression evaluator into the second code segment, if `_DUALCODE` is also enabled. This moves 5 kB out of the first code segment. Performance may suffer from this option. Disabled by default.

`_SYMBOLASMDUALCODE`

Moves most code of the `symbols.asm` file into the second code segment, if `_DUALCODE` is also enabled. Disabled by default.

`_CHECKSECTION`

Sets default value for the following three options. Disabled by default.

`_CHECKCALLSECTION`

`_CHECKJMPSECTION`

`_CHECKJCCSECTION`

These options enable detection of invalid cross-section branches. Enabling all of these options makes the build take a long time, possibly in excess of two and a half minutes (versus 30 seconds). Further, memory use of the assembler may as much as triple easily. (166 MiB seen, versus 66 MiB.) The patch for the %rep leak for NASM versions beyond 2.16.01 is a must for enabling any of these options. These options are not useful if the `_DUALCODE` option is not enabled.

`_EXTENSIONS`

Reserves memory for Extension for lDebug loading and adds the EXT command to load. (DOS must be available for application mode or device mode. Auxiliary buffer must be available for bootloaded mode.) The `/X=` switch to the application mode or device mode debugger can be used to set the size of the ELD instance code segment. Likewise, the `/Y=` switch can be used to set the size of the ELD data area. Enabled by default.

Section 4: Getting started with the release

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

In the `bin` or `BIN` directory:

`ldebug.com`

Compressed bootable debugger, build without DPMI support

`ldebugx.com`

Compressed bootable debugger, build with DPMI support

`instsect.com`

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

`*.eld`

Extensions for `lDebug`

Only in the stand-alone package, `bin` also contains:

`ldebugu.com`

Uncompressed bootable debugger, build without DPMI support

`ldebugxu.com`

Uncompressed bootable debugger, build with DPMI support

In the FreeDOS package, `SOURCE/LDEBUG/ldebug/bin` contains all the same files as `bin` in the stand-alone package. (All files in `BIN` are duplicated from this `bin` directory.) Likewise, the stand-alone package's `tmp`, `doc`, and `lst` are carried over into the FreeDOS package's source tree.

Unlike the stand-alone package, the FreeDOS package contains all sources needed to build the debugger from scratch, except for the build toolchain. The following repos' most recent tip revisions are copied into the `SOURCE/LDEBUG` source tree:

`inicomp`

Depacker stages for compressed triple-mode executables

`instsect`

Install boot sector loaders to drives or file system images

ldosboot

Boot sector loaders and initial loader stage for kernels (triple-mode executables supported)

lmacros

IDOS assembly macro collection

scanptab

Allow initial loader or bootloaded debugger to scan for partitions

tellsiz

Display the allocation size needed by a .big style application image

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 (lzsa2) is chosen based on the executable size and depacker performance. Other compression method choices may be desired. 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 (with CR LF line endings)

ldebug.pdf

Manual in PDF (only in stand-alone package or source tree)

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.map`

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

`debugx.map`

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

`instsect.map`

Assembly map corresponding to `instsect.com`

`boot12.map`, `boot16.map`, `boot32.map`

Assembly maps corresponding to the loaders that are embedded into `instsect.com`

Listing files are no longer shipped with release builds (as of release 8), neither in the stand-alone package nor the FreeDOS package. To obtain a listing file that will match the binaries of a release build, one may obtain the daily current build of the same day as the release build. The only differences should be in the debugger version strings, which are filled to the same size to avoid changing the offsets of everything.

Alternatively, in the source directory `./make reproduce` may be run on a host system to recreate all artefacts of the build of the main debugger. (Exact identicalised output may require the same toolchain as originally used.) Likewise, in the source directory `./makinst.sh` may be run to recreate the `instsect` artefacts and in the `source/eld` directory `./mak.sh` to create the ELDs.

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 EDR-DOS, FreeDOS, RxDOS.3, and IDOS 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 10.25), 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 10.58). The filename is however `LDDEBUG.SLD` for DDebug builds, and `LCDEBUG.SLD` for CDebug builds.

Executing the Q command (section 10.34) makes the debugger uninstall itself then continue running whatever code the debuggee is in. Executing the `BOOT QUIT` command (section 18.11) 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 17.1 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.

`/IN`

Clear the command line buffer. This switch can be used to discard the default commands

placed into the command line buffer. They are intended to load a start up configuration from a Script for IDebug file, either in the directory specified by the %LDEBUGCONFIG% variable or in the debugger executable's directory.

/A

Specify auxiliary buffer size. The size may be specified as the keywords MIN, MAX, a hexadecimal number, or a hash sign # followed by a decimal number. If a blank follows directly behind /A this is parsed like /A=MAX. Minimum size is 8208 Bytes, maximum size is by default 24576 Bytes. The auxiliary buffer currently cannot be resized by the resident debugger. This switch is processed by the debugger's init.

/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. Refer to section 10.59.1.

/B

Run a breakpoint within the debugger's initialisation.

/F

Enable/disable treating file as a flat binary. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This controls the DCO6 option 400h. If enabled, .EXE and .COM files will be loaded as flat binaries even if they contain an MZ executable header. .HEX files will also be loaded as flat binaries rather than being interpreted to load the described contents. Writing the files back as flat binaries is also enabled by this. (Note that the file has to fit into memory for this.) /F implies /E+, but /F+ and /F- do not imply anything about /E.

/E

Enable/disable setting Stack Segment != PSP for loading flat binaries. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This controls the DCO6 option 800h. If enabled then loading a flat binary file (with filename extensions such as .BIN or using the /F switch) will set up a Stack Segment at the end of the process memory block. That is, in a different segment than the process segment. If disabled, then flat binaries always get SS = PSP even if that leaves the stack pointing into the binary image. /F implies /E+.

/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 10.54.

/2

Enable/disable using a second display for debugger output. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This option is disabled by default. If no second display is detected by the debugger, trying to enable this mode causes an error message. This option can only be enabled by the debugger's init. Therefore there is no Debugger Common Options flag for this mode.

/P

Enable/disable path search and guessing filename extension. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This option is disabled by default. When enabled, the debugger's init will try to expand the filename passed after the switches. (This only works during initially loading the debugger. Subsequent N or K commands will not observe /P+ mode. Therefore there is no Debugger Common Options flag for this mode.)

If the last component of the filename (after the right-most forward slash, backslash, or colon) does not yet contain a dot, the debugger will first try to load from the filename as given. Then it will try to append three different filename extensions in order: .COM, .EXE, and .BIN.

If the filename does not contain any path components (indicated by forward slashes, backslashes, or colons) and no match is found in the current directory, the debugger will read the %PATH% variable if any. It will then attempt to find a match in every path element in order. (If the filename does not contain a dot, then for every path element, it is searched for the filename as-is and then with each of the three extensions added. If all four attempts fail, the next path element is tried.)

/PE

Enable/disable guessing filename extension, only.

/PS

Enable/disable path search, only.

/PW

Enable/disable warning for unknown filename extension. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This option is enabled by default. When enabled, the debugger's init will match the filename passed after the switches to a list of known extensions. (This only works during initially loading the debugger. Subsequent N or K commands will not observe /PW+ mode. Therefore there is no Debugger Common Options flag for this mode.)

If the filename is not empty, and if the last component of the filename (after the right-most forward slash, backslash, or colon) does contain a dot, and the last four text bytes do not match a known extension, then the warning is displayed by init. The known filename extensions are: .HEX, .ROM, .COM, .EXE, and .BIN.

/D

This switch is only used for a conditionally debuggable build (ICDebug). Enable/disable debuggable mode. Enable if a blank or plus sign follows this switch. Disable if a minus sign follows this switch. This controls the DCO6 option 100h. By default, debuggable mode is enabled. For IDebug or IDDebug builds (not ICDebug), a no-op /D switch is accepted but a switch that disagrees with the debuggability of the build is rejected.

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 10.34) 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 10.51) 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.

The debugger detects its configuration directory as follows:

1. If the environment variable LDEBUGCONFIG is set, read it.
2. Else, if the path to the debugger's executable can be determined, use that.
3. Else, the current directory on the current drive is used.

The command line buffer receives a command that uses the configuration directory. This command is written before any /C= switch contents.

It is equivalent to checking for a file and label using the IF command (section 10.25), then if found to execute that script file. The IF condition is like `if exists y ::config::LDEBUG.SLD :applicationstartup` then and the subsequent script command is `y ::config::LDEBUG.SLD :applicationstartup` (section 10.58). The filename is however LDEBUG.SLD for DDebug builds, and LCDEBUG.SLD for CDebug builds. This default command can be disabled using the /IN switch.

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. (The /T switch is the exception, it is only valid to the application mode.) In particular, /C= can be used to pass commands to execute. The configuration Script for lDebug file is detected in the same way as for application mode, except the label used is :devicestartup.

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. (Refer to the Test Reference in section 20 for a list of all tests.)

The script uses the following environment variables:

build_name

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

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. This should include the command 'r dco6 clr= 100' or 'uninstall debug' if testing ICDebug to disable its debuggable mode.

test_sleepduration

Floating point number which defines the default sleep duration, in seconds, for read calls that do not override it. This defaults to 0.05.

test_addsleepduration

Floating point number which defines a duration, in seconds, to add to the duration of overridden read calls. This defaults to 0.0.

DEFAULT_MACHINE

‘qemu’ or ‘dosemu’ (default is ‘qemu’)

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.

The most common reason for random failures is timing. If this is suspected to be the case, the duration variables allow increasing the time spent waiting on debugger output. They were added to replace the workflow of editing durations manually in the test script.

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, `INSTALL BIOSOUTPUT` has been used, 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 or device mode debugger (not bootloaded) while DOS's InDOS flag is set. (This check always uses the actual InDOS flag, ignoring the "force InDOS" mode of the debugger.) A tilde ‘~’ is prepended for DDebug, or CDebug while in debuggable mode.

If DOS command line input is done using `getinput` (eg if DCO option 800h is set or `INSTALL GETINPUT` was run) 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. To discard the current line and re-enable history recall, Control-C may be entered.

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 12.11 for the serial configuration variables. Setting the DCO flag 4000h enables serial I/O. This flag can also be set using the `INSTALL SERIAL` command. 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, DDebugX, CDebug, or CDebugX. The prompt indicator is ‘~= ’ for DDebug or CDebug while in debuggable mode.) 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. The flag can also be cleared using the UNINSTALL SERIAL command.

6.4 Register dumping

The R command (refer to section 10.37) 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 10.37.1) 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 10.49, section 10.49.1, section 10.33.) It also includes a run with the G command. (Section 10.21.) Further, a permanent breakpoint which is configured as a pass point being passed also runs the RE buffer commands. (Section 10.7.)

Setting the flags 1_0000 or 4_0000 in the DCO3 variable enables register change highlighting. (The command INSTALL RHIGHLIGHT sets DCO3 4_0000.) 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 2_0000 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. (Recent dosemu2 in -dumb terminal mode is detected as not being functional. Current dosemu2 is queried for -dumb mode directly, and considered not functional if in this mode.)

If this check fails or the DCO3 flag 2_0000 is set then escape sequences are written using interrupt 10h.

6.5 Memory dumping

Another basic command is the D command (section 10.12). 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  Gthv1w2r./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 Loading the debuggee

A program to examine can be loaded using the N and L commands. If the debugger is loaded as a DOS application with a filename specified in its command line, it will run the N and L commands on its own.

The N command sets up some buffers internal to the debugger. One of those specifies the pathname of the executable file to load. The pathname must include the filename extension, if any. The pathname must be relative to the current directories at the time the L command runs, or it must be absolute. This is not true of the pathname initially specified on the debugger command line tail if the debugger switch /P is used. The path Extension for lDebug (refer to section 15.41) can be loaded to provide extension guessing and path search for later N or K commands after the debugger has been loaded properly.

The tail of the N command after the pathname is used as the command line tail for a new debuggee process.

The L command without any parameters attempts to load the program specified to the last N command into a new process. If the L command does not display any messages this indicates success.

6.8 Running the debuggee

Once a program is loaded into the debugger it can be run in several ways:

G command

Runs at full speed until a breakpoint is encountered. Temporary breakpoints can be specified to the G command. Refer to section 10.21.

T command

Traces a single instruction, except for software interrupts which are by default run at full speed with a breakpoint after them. Refer to section 10.49.

P command

Either runs at full speed with a breakpoint behind the current instruction, or traces a single instruction. Software interrupts, call instructions, repeated string instructions, and loop instructions are proceeded past by using a breakpoint. Refer to section 10.33.

TP command

Like the T command except that repeated string instructions are proceeded past like the P command would. Refer to section 10.49.1.

All run commands support auto-repeat: Submitting an empty line to the debugger (blanks allowed but no comment) will make the debugger run the last command again. For the G command auto-repeat, the specified temporary breakpoints will be used again. Refer to section 10.1.

Permanent breakpoints can be set up and changed using the B commands. They can be configured to behave as pass points as well. Refer to section 10.7.

The ?RUN help page in section 18.8 lists some additional features of the T, P, and TP commands.

6.9 Help

The online help can be accessed using the ‘?’ command. Refer to section 18 for copies of the online help.

Section 7: Debugging the debugger itself

There are debuggable builds of the debugger, called IDDebug (unconditionally debuggable) and ICDebug (conditionally debuggable).

The debuggable mode works by installing the mandatory interrupt handlers of the debuggable debugger only within the 'run' function, so as to return the control flow to this instance when it runs its debuggee code. On return into this instance, it uninstalls its mandatory handlers again. This mechanism allows to debug most of the debugger using a different instance of IDebug (or potentially another debugger).

In debuggable mode, an additional command is supported, the BU command (which stands for "Break Upwards"). It will run a breakpoint within the debugger's code segment which will break into the other debugger. Its code was updated so it will break at the command dispatcher after the label cmd4. This means if the outer debugger is also an IDebug then it can be instructed to skip to the next command being dispatched by entering the command 'G ip'.

IDDebugX (or ICDebugX) can also install its exception areas into the other IDebugX instance. For this, the other debugger needs to have run an 'INSTALL AMIS' command. Then the debuggable debugger can run its 'INSTALL AREAS'. Afterwards, faults in the debuggable debugger will make the other IDebugX indicate the area of the fault.

After IDebugX has caught a fault in the CODE or CODE2 segment, it can be instructed to resume the IDDebugX (or ICDebugX) command input loop (cmd3) by running a 'G=0' command. If 'G=1' is used instead, an additional linebreak will be displayed by the debuggable debugger before it starts prompting for input. This is useful if the fault occurred with some partial output currently displayed. The offset 0 and offset 1 entries are also supported by non-DPMI builds and can of course be used at any point in time other than after a fault, too.

There are some DCO6 flags to control breakpoints and entering ICDebug's debuggable mode in the functions debuggerexception and putrunint. They can be displayed using a '?O6' command.

Other than for the most trivial sessions it is recommended to control the outer debugger by serial I/O, separately from the I/O of the debuggable debugger. If the latter also should be controlled by serial I/O then two different ports can be used. The terminal connected to the outer debugger can also be set up for TracList, the IDebug companion application which traces a listing file. For instance, if IDDebugX is to be traced, TracList should be run with the ldebug.lst/ddebugx.lst listing file.

7.1 Initialising the debuggable debugger

To allow the debuggable debugger to relocate and initialise its code sections, the outer debugger should generally start running the debuggable debugger with a plain 'G' command. The debuggable debugger can then return control to the outer debugger using its 'BU' command.

If the initialisation of the debuggable debugger is to be debugged, the '/B' switch may be of use.

Otherwise, note that the NEC V20/V30 and 486 CPU detections may fail when traced using an outer lDebug.

The NEC detection may lock the machine up if its specially encoded 'pop cx' is traced or run with a breakpoint directly behind it. To allow to continue tracing after it, a breakpoint must be set up at the 'jcxz' instruction or later. There must not be a breakpoint on the 'mov sp, ax' instruction. The 'pop cx' instruction must not be traced with the Trace Flag set. Failure to honour these requirements may lock up the NEC CPUs, for example the one used in the HP 95LX, which then may require resetting the system with Ctrl-Shift-On. This also resets the system date and time.

The 486 detection may wrongly detect a 386 instead of a 486+ when traced on some systems, such as some revisions of dosemu(2).

7.2 Sectioning overview

The sections of the debugger are declared in debug.asm:

```
org 100h
addsection lDEBUG_DATA_ENTRY, align=16 start=100h
data_entry_start:
#define DATASECTIONFIXUP -data_entry_start+100h
_CURRENT_SECTION %+ _start:
%xdefine %[_CURRENT_SECTION %+ FIXUP] - _CURRENT_SECTION %+ _start+100h

addsection ASMTABLE1, align=16 follows=lDEBUG_DATA_ENTRY
addsection ASMTABLE2, align=16 follows=ASMTABLE1
addsection MESSAGESEGMENT, align=16 follows=ASMTABLE2 vstart=0
messagesegment_start:
addsection lDEBUG_CODE, align=16 follows=MESSAGESEGMENT vstart=0
code_start:
#define CODESECTIONFIXUP -code_start+0
_CURRENT_SECTION %+ _start:
%xdefine %[_CURRENT_SECTION %+ FIXUP] - _CURRENT_SECTION %+ _start+0
addsection lDEBUG_CODE2, align=16 follows=lDEBUG_CODE vstart=0
code2_start:
#define CODE2SECTIONFIXUP -code2_start+0
_CURRENT_SECTION %+ _start:
%xdefine %[_CURRENT_SECTION %+ FIXUP] - _CURRENT_SECTION %+ _start+0

addsection DATASTACK, align=16 follows=ASMTABLE2 nobits
addsection INIT, align=16 follows=lDEBUG_CODE2 vstart=0
%if _DEVICE
addsection DEVICESHIM, align=16 follows=INIT vstart=0
%endif
addsection RELOCATEDZERO, vstart=0 nobits
relocatedzero:
```

These are the sections:

lDEBUG_DATA_ENTRY

Data entry section. Used for data, most messages, and code entrypoints into the debugger.

Addressed with the same segment as the debugger's PSP. This segment is usually addressed using SS, often DS also, and sometimes ES too. Not relocated, except if application /T switch used.

ASMTABLE1, ASMTABLE2

Assembler/disassembler tables. Addressed with the same segment as the debugger's PSP and the data entry section. Not relocated, except if application /T switch used.

MESSAGESEGMENT

Used to store some long messages, as well as the ELD link info tables. Addressed with a vstart of zero using its own segment. Relocated to behind the data stack section when the init sets up the debugger. May be truncated to drop ?BOOT help page when not in bootloaded mode.

IDEBUG_CODE

First and main code section. Addressed with a vstart of zero using its own segment. This segment is addressed using CS most of the time. Relocated to behind the message segment or behind the auxiliary buffer. May be truncated to drop boot code when not in bootloaded mode.

IDEBUG_CODE2

Second code section. Only used (not empty) if _DUALCODE is enabled. Addressed with a vstart of zero. Relocated to behind the first code section.

DATASTACK

Nobits. Uninitialised data and stack section. Addressed using the same segment as the data entry section. This contains the ELD data blocks, the size of which can be changed using the /Y= switch to the application or device driver init. The maximum ELD data size will expand this section up to offset 65_520.

INIT

Debugger init. Addressed using a vstart of zero. In device mode and application mode, this section is relocated at least once. A second relocation will be attempted if _INIT_MAX=0 and the chosen sizes of the auxiliary buffer, the ELD code segment, the history buffer, and the ELD data blocks are too large to fit into the initial allocation. Discarded after init is done.

DEVICESHIM

This section is eventually installed at the beginning of the debugger allocation for the device mode debugger, before the debugger's pseudo-MCB and PSP created by the init. It contains the header and resident code of the device driver that the debugger installs to stay resident.

RELOCATEDZERO

Nobits. A label in this section is used in far call (dualcall with _PM=0) instructions for the segment immediate. Its use is to emit a relocation into the listing file, helping these instructions to be matched by TracList.

There are some additional segments that do not correspond to sections in the debugger's sources. All of these segments are allocated at init time.

Auxiliary buffer

Used for various needs. Default size is 8 KiB + 16 Bytes, can grow up to 24 KiB using /A= switch to non-boot init.

History segment

Used to hold line editor history. Default size is 8 KiB, can grow up to nearly 64 KiB using /H= switch to non-boot init. Can shrink to nearly 260 Bytes as well.

Environment block

Holds the debugger's environment. Size is 2 KiB.

ELD code segment

Provides space for loading ELD code instances. Default size is 16 KiB, can grow up to 65_520 Bytes or shrink all the way to zero bytes using the /X= switch to non-boot init.

Some additional memory allocations show up as gaps in the memory map of the debugger:

Device driver header and code

This is initialised from the DEVICESHIM section.

Device driver trailer paragraph

This is reserved to support the QC command, in case the debugger is stored in an SD sub-MCB.

Layout 3 gap

When the debugger init determines that the auxiliary buffer cannot be placed as desired in one of the first two layouts, then a gap of about 8 KiB is allocated additionally.

Boot image ident

This paragraph contains a signature that spells 'NDEB' and size information on the allocation of the debugger.

Boot alignment gap

The boot allocation is done using the int 12h memory size. This size is specified in an amount of kilo binary bytes. Therefore, the debugger may need to allocate up to 1008 Bytes more than it needs to align its allocation to a KiB boundary.

7.3 Debugging ELDs

The ELD code segment has a different convenience entrypoint that returns the control flow to the cmd3 loop. Due to the structure of ELD instances, there can be no code at offset 0. Instead, there is an entry at offset 79h. Therefore, when the inner debugger is running code in the ELD code segment, a 'G=79' command can be used to cancel the currently running ELD. (At most points in an ELD, to cancel the currently running ELD should be safe. In particular, any calls to debugger code that may do I/O or that may invoke the error handler must be safely cancellable.)

7.3.1 Using the offset hint to debug ELDs

ELDs can be loaded at arbitrary offsets in the ELD code segment. The first ELD is always loaded at offset 80h, but subsequent ELDs may be loaded at higher offsets. The only certainty is that the offset is paragraph-aligned.

To help in using TracList to debug an ELD, a special handshake is provided to communicate an ELD's offset.

The outer debugger should `'install amis'` and install the AMIS message ELD by running `'ext amismsg.eld install'`. When the ELD linker runs in the inner debugger, it will send a hint message to the outer debugger's AMIS function 40h (refer to section 13.5.5). The hint is received by the AMIS message ELD of the outer debugger and displayed to the outer debugger's terminal before processing the next command in the `cmd3` loop.

The hint line looks like this example:

```
TracList-add-offset=ldmem.lst::0080h
```

When the hint line is received by `tractest`, it will hand it over into the line file and hint file, both read by TracList. TracList will then parse the offset provided by the hint. In the current use case this hint is always used to add a file-scoped offset.

7.3.1.1 Using the hint ELDs

There are additional ELDs, `hint.eld` and `hintoth.eld`. Both will enumerate listing hints for all currently loaded ELDs.

The first, `hint.eld`, will send a string containing all the hints to the AMIS message service of another debugger instance.

The second, `hintoth.eld`, will display the hints of an other link debugger to the terminal of the debugger instance that runs the `hintoth.eld`. This requires the other debugger instance to have installed its AMIS interface and loaded the `amisoth.eld` to provide the other link.

7.3.2 Using houdinis to debug ELDs

Houdinis are conditional breakpoints. They run an `int 3` instruction only if three conditions are met:

1. Houdinis are enabled in the ELD's build
2. The debugger is in debuggable mode (for `ICDebug`) or it is an unconditionally debuggable build (`IDDebug`)
3. Houdinis have been enabled in the DCO7 using eg `'install houdini'`

7.3.3 ELD examples

`ldmem` - Command hook

Depicts how to install and uninstall a resident ELD with a command hook.

`ldmem` - Nouns

Dispatches to subfunctions using a table of nouns that can be entered on the command line,

defaulting to ALL if no nouns are specified.

ldmemoth - Other linker use

Uses the other link info to access data structures of a different instance of lDebug running on the system. (Almost all of the sources are shared with ldmem.)

amismsg - AMIS handler

Implements two TSR-private AMIS services, functions 40h and 41h, by using the AMIS handler hook.

amisoth - Export link info

Exports the link info and data/entry/stack segment and selector of the debugger on the AMIS interface, to be read by ldmemoth.

aformat - Iterate a table of handlers

This ELD hooks into a record 4 handler chains exposed by the debugger. To handle all of them, the install and uninstall code iterates over a table that points to the handlers and the hooks in which they are referenced.

list - Find files with wildcard pattern

The list ELD finds and opens files specified by a pathname pattern. It can use DOS LFN find, DOS SFN find, or a bespoke boot directory scan to match filenames.

list - Use eldstrict mode with annotated relocations

This mode utilises the recent NASM's reloc warnings to enforce relocations being marked by the reloc or reloc2 mmacros.

rn - Do 386-related patches upon loading the ELD

The RN ELD implements patching based on whether it is running on a 386+ machine or not. It re-uses the debugger's patch and table writing macros.

amount - Install an ELD variable

This installs ELDAMOUNT as a variable that can be read by the debugger's expression evaluator.

eldcomp - Modify output in the puts handler

When detecting a difference, eldcomp will rerun the ELD under test and allocate buffers to copy the ELD's allocation. Then it will run a debugger C command to highlight differences. To relate the C command output to the ELD's listing file offsets, eldcomp hooks into the puts handler and writes the in-section offset before and after the C command output.

eldcomp - Inject debugger commands

To run its test commands, eldcomp installs itself as a resident ELD and hooks into the debugger's command injection handler. It then injects up to 17 commands to be run by the debugger.

ifext - Add an IF command

When installed as a resident ELD, ifext will provide the IF [NOT] EXT extensionname THEN command construct.

x - Add a help page to the ? command

The EMS commands ELD will add its help page using either of two commands: X? or ?X.

set - Redirect output in a puts handler

When run with the /E switch, the set ELD will inject a command with a puts handler installed. This handler will save one line of output and cause none of the output to actually be displayed.

withhdr - Provide prefix commands

The WITH HEADER and WITH TRAILER commands are prefixes that modify the workings of a subsequent command.

variable - Preprocess command text

This ELD installs a preprocess handler to expand DOS environment variables in commands.

dpb - Get an address that is always a 86 Mode segmented address

Uses getaddrX to get an 86 Mode segmented address, even in Protected Mode. This happens to work. The address is read if the DPB command is used with the LIST AT keywords.

Section 8: Parameter Reference

8.1 Number

Plain numbers are evaluated as expressions. Refer to section 9. 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'`.

8.2 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.

Instead of an address, the address parameter may consist of the taken keywords: `TAKEN` or `T` for taken, and `NOTTAKEN` or `NT` for not taken. This is only valid if the current `cs : (e) ip` points at a conditional branch instruction, and will cause a parsing error otherwise. The taken keywords will evaluate to a segmented address pointing at the target of the conditional branch. The not taken keywords will evaluate to a segmented address pointing to behind the conditional branch instruction.

Address parameters are used by a lot of commands.

8.3 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', 'DWORDS', 'QWORDS', 'PARAS', 'PARAGRAPHS', 'PAGES', 'KiB', 'MiB', or 'GiB'. Except for the first, the plain number will be shifted as for the specified unit size (multiplying by 2, 4, 8, 16, 512, 1024, 1_048_576, or 1_073_741_824). It is an error if the shifting overflows the debugger's 32-bit arithmetic. 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 $4 \times 4 = 16$ bytes. The item size keywords were introduced primarily for the 'DW' and 'DD' commands (refer to section 10.12), but they can be used for any command that accepts a range.

There is a new keyword called 'END'. This keyword may appear where the 'L' or 'LENGTH' keyword would be expected. After the 'END' keyword the end offset is parsed. This is the same behaviour as if no keyword was present but crucially it allows an end offset starting with the text 'L', which would otherwise be misparsed as an 'L' keyword.

If the default length is used (the line ends after the start address) then a start address near the end of a segment (1_0000h or 1_0000_0000h) will shorten the length if it would otherwise overflow the segment.

Range parameters are used by a lot of commands.

8.4 Range with **LINES** keyword allowed

This type of parameter is an extension of the range parameter type. Both the default length and the explicit length may be specified as a number of lines instead of an address length.

An explicit 'LINES' length is specified by prepending an 'L' or 'LENGTH' keyword (like an address length) but then specifying a unit as 'LINES' instead. The number of lines specified must be nonzero and below 8000h.

The exact details of how a lines length is used depend on the command in question. A range with lines length is allowed for the U command (section 10.52) and the D/DB/DW/DD commands (section 10.12).

8.5 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 10.18, section 10.20, and section 10.47.

A list may have its type changed with an AS keyword, followed by a size keyword BYTES, WORDS, or DWORDS. When a larger size is selected, each subsequent number expression and

each byte of quoted text are written to a full word or a full dword instead of to a byte. Text bytes are zero-extended. Numbers can calculate to any value fitting the specified size. A list's type may be changed multiple times within the list.

8.6 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 10.47 and section 10.20.

8.7 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.

8.8 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 10.7.

8.9 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 10.14 and section 18.11.

8.10 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 10.7 and section 10.21.

8.11 Label

A label is a (not quoted) string keyword. A label can be used by the GOTO and Y commands, refer to section 10.22 and section 10.58. For the Y commands a label must start with a colon. For the GOTO command the colon may be specified but it is optional.

8.12 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 10.24 and section 10.32.

8.13 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 10.28 and section 10.56. The EXT, N, and Y commands (section 10.19, section 10.31, and section 10.58) also accept drive parameters, but only as part of their filenames. These must be in the drive letter followed by colon format.

8.14 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 10.28 and section 10.56. Some BOOT commands also use sector numbers, refer to section 18.11.

8.15 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 10.25, section 10.33, section 10.49, section 10.7.3, section 10.7.1. 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.

8.16 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 10.37.

8.17 Command

Command is a special parameter type that is used only by the RE.APPEND, RE.REPLACE, RC.APPEND, and RC.REPLACE commands (section 10.37.2 and section 10.37.4). 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.

8.18 ID

ID is a special parameter type that is used only by the BP and BI commands (section 10.7.1 and section 10.7.2). 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.

8.19 Filename or pathname

This parameter type is used by EXT, N, K, and Y commands, as well as some BOOT commands. EXT and Y commands allow to use double quote marks. When using DOS, EXT and Y commands can access files using Long File Names (LFNs). When using DOS, all available commands parsing filenames may specify drive letters. EXT and Y commands when bootloaded, and some BOOT commands, may specify partitions at the beginning of filenames.

8.20 Command line tail

Command line tails are parsed by EXT, N, and K commands. They always are located behind a

filename parameter. A command line tail may be empty. The N and K commands will store the command line tail for use by the L program-loading command. The EXT command passes its command line tail to the Extension for lDebug that it loads. The ELD may parse its command line tail in whatever way is desired, which may involve parsing other parameter types.

Section 9: Expression Reference

9.1 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, inclusive. Multiple hash signs are allowed in the same literal.

9.2 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.

9.3 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.

9.4 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.

9.5 Parentheses

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

9.6 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.

9.7 **DESCTYPE** keyword

This keyword introduces a descriptor type read. The following expression is taken to be a selector specification. This keyword is only valid for (DPMI-enabled) IDebugX builds, and only while in Protected Mode.

The value is read from a 'lar' instruction on the following expression, and shifted to the right by 8. If the instruction indicates that the selector does not refer to a valid descriptor then the result of this keyword is zero.

9.8 **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.

9.8.1 **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:cip LENGTH abo - cip
```

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 12.19).

WRITING

Expands to a comma-separated list of `FROM writadr0 LENGTH writlen0` constructs, for every write access variable pair (refer to section 12.19).

ACCESSING

Expands to `READING, WRITING, EXECUTING`.

9.9 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.

Any side effects that may happen from parsing and reading the second operand or the third operand will always happen, even if the operand in question is not selected as the result by the construct.

9.10 Expression side effects

Some uses of the expression evaluator may have side effects. These side effects may happen even if the parsing of an expression or a command ultimately fails. As a special case, side effects may occur up to twice if a machine mode command (section 10.30) is parsed.

The ternary `?? ::` operator and the `VALUE IN` construct will both always evaluate every operand that they're given, even if that operand is not selected as the result or does not contribute to the match count.

Possible side effects include:

- LFSR and RLFSR variables will be stepped once each time they're read.
- Indirection can read access arbitrary memory in the current mode, making it possible to affect memory-mapped I/O if such memory is visible to the debugger. Other variables may also read memory, but not as arbitrary as indirection.
- If an address parsed within an address parameter or in indirection or in a `LINEAR` construct includes a dollar sign prefixed segment or pointer type expression, then `IDebugX` may request a selector from the DPMI host.
- If the symbolic build option is enabled, symbol table access in XMS or 86 Mode memory may occur.

Section 10: Command Reference

10.1 Empty command - Autorepeat

Entering an empty command at an interactive prompt results in autorepeat. Empty means no content except for blanks. A line starting with a semicolon comment is not considered empty. 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 element size as the prior dump. As for if the command is executed with an address lacking a length, the default length is used. (It defaults to 128 bytes, refer to section 12.5.)

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 12.4.)

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 12.4.)

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 12.4.)

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 is used. (It defaults to 32 bytes, refer to section 12.5.)

10.2 ? 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 pages	?OPTIONS
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 18.

10.3 : prefix - GOTO label

A leading colon indicates a destination label for GOTO, see section 10.22.

10.4 . (dot) command - Immediate assembler

A dot command can be used to invoke the immediate assembler. This is only available if the `_IMMASM` build option was enabled. Following the dot, an instruction is parsed which is assembled. If successful, then the assembled instruction is immediately run.

Branches and instructions involving CS as a prefix or operand are handled specifically to do something equivalent to the assembled instruction, by a combination of special detection and modification or as-if handling. This includes `jmp far/near/short`, `jcc`, `loop`, `call far/near`, `retf/retn/iret`, `mov to ds`, `mov from ds or cs`, `push cs`, `push ds`, `pop ds`, `lds`, and all memory operands with cs prefix. (The instructions involving ds are handled specifically because a cs prefix is replaced by a ds prefix and ds is temporarily replaced by the cs value then.)

Interrupt calls are always proceeded past, even if TM is set. The `int` instruction is run from a buffer internal to the debugger. Interrupts which depend on the CS or (E)IP they're called from may not work as expected. Calls are usually traced, but can be proceeded past by including a comma after the dot command.

Warning: It is generally not safe to modify SS or SP to relocate the stack with the immediate assembler. This will fail because the immediate assembler traces most of the instructions assembled with it, which uses the stack both before and after running the instruction. LSS SP or LSS ESP are okay if the stack is valid immediately after the instruction is traced. To relocate the stack otherwise you may use the R command to modify the SS and (E)SP debugger variables. This does not trace anything in between modifying the two variables.

10.5 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, wherever 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.

10.6 ATTACH command - Attach to process (Leave TSR mode)

```
attach process    ATTACH psp
```

While in resident mode, the ATTACH command can be used to attach the debugger to a process. This is the opposite operation to the TSR command (see section 10.51). The device driver mode starts out as detached while the application mode starts out as attached.

The provided parameter must be a segment value, even if IDebugX is in Protected Mode. It refers to the PSP to attach to. This PSP must not be self-owned.

To attach to a process, the debugger stores away the current PRA and parent of the process, and modifies both to point to the debugger instead.

When attached to a process, commands like QA and the process-loading L command can be used sensibly. The Q command will also try to terminate the attached process.

When detached, the Q command will continue to run the current debuggee context after the debugger has been uninstalled.

The usual parameter to the attach command consists of the 'PSP' variable, which will have the command try to attach to the current debuggee process. Other choices like ATTACH PARENT are also valid.

10.7 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.

Example counter values:

8000h (default)

Always break

4000h

Always pass

8003h

Break on third time the breakpoint is reached, then break always

0003h

Break on third time the breakpoint is reached, but do not break again

C006h

Pass for five times, then on the sixth time the breakpoint is reached break on it, then break always

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.

10.7.1 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, if any, is reset (unlike the NEW keyword), 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 10.7.4 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.

10.7.2 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.

10.7.3 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.

10.7.4 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.)

10.7.5 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.

10.7.6 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.

10.7.7 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.

10.7.8 BE command - Enable breakpoint

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

Like BD, but enables breakpoints.

10.7.9 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.

10.7.10 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.

10.7.11 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
-
```

10.8 BU command - Break Upwards

break upwards BU

This command, which is only supported by Debuggable builds (DDebug) or Conditionally Debuggable builds (CDebug), causes the debugger to execute an int3 instruction in its own code segment (IDEBUG_CODE). This breaks to the next debugger that was installed prior to DDebug or CDebug. Prior to the breakpoint, the message "Breaking to next instance." is displayed. The breakpoint is in the cmd4 dispatcher. When the next instance is another lDebug then running the command 'G ip' in it can be used to run the debuggable debugger until the next command is dispatched. (This does not work if an Extension for lDebug command handler processes a command instead of passing it on to the debugger.)

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

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

In conditionally debuggable builds, the following message is displayed instead if CDebug is currently not in debuggable mode:

```
-bu
Debuggable mode is disabled.
Enable with this command: r DC06 or= 0100
-
```

10.9 BOOT commands - Boot loading support

The BOOT commands are only available if the debugger is running in boot loaded mode.

10.9.1 BOOT PROTOCOL= command

```
BOOT PROTOCOL=proto [parameters] [partition] [pathnames [cmdline]]
```

This command is used to load a boot sector or kernel using the loaders implemented by the debugger. These loaders attempt to be highly compatible to the original loaders whose load protocols they simulate.

10.9.1.1 Specify protocol

Using the keyword `PROTOCOL`, the load protocol to use as a base can be specified. This keyword is required, unless the special protocol named `SECTOR` is to be used.

10.9.1.2 Altering protocol parameters

When specifying a protocol other than the special `SECTOR` protocol, the protocol parameters can be altered. Each protocol will set up defaults for all of those parameters. Each protocol can be completely described by a combination of parameters and default filenames. Every parameter is indicated by a keyword followed by a numeric expression, or in some cases followed by a segmented address.

The following parameters are available:

MINPARA

Specify minimum amount of paragraphs to load from the first file. It is an error if a file is shorter than that.

MAXPARA

Specify maximum amount of paragraphs to load from the first file. It is valid for the file to be shorter or longer than this. If nonzero then it is an error if the file is so long that there is not enough memory to hold this amount of paragraphs. If zero, then as much of the file is loaded as fits.

SEGMENT

Specify load address of the data from the first file. The number specified is taken to be the segment of an address within memory.

ENTRY

Specify entrypoint to set up in the CS:IP registers. If a single numeric expression, it is taken as the offset (for IP) and the segment value is assumed as zero. That is, CS will be set up to equal the `SEGMENT` parameter in use. If a segmented address, the offset is used for IP and the segment is used as a relative adjustment to the `SEGMENT` that is in use to obtain the value for CS. It is valid for the segment value to be positive or negative.

BPB

Specify where to load the boot sector with (E)BPB. If a single numeric expression, it is

taken as the offset in segment zero. If the segment is specified as 0FFFFh or -1, then the "auto-BPB" feature is used and the boot sector and stack is located at a high address that is not otherwise used. The offset is still set to the offset part in this case.

CHECKOFFSET

Specify offset of word value to check. Must not possibly cross a sector boundary. (This is checked by testing that the offset modulo 32 is not equal to 31.) May not be higher than 0FFFEh.

CHECKVALUE

Specify value of word to check. If zero, no check occurs. It is an error if this value is nonzero and the check does not match.

The following boolean parameters are available. Like the other parameters they read a numeric expression, but this is only checked to be true (non-zero) or false (zero).

SET_DL_UNIT

If true, set up the DL register with the load unit. This is used by several protocols.

SET_BL_UNIT

If true, set up the BL register with the load unit. This is used by the FreeDOS and EDR-DOS protocols.

SET_SIDI_CLUSTER

If true, initialise the DI (FAT12/FAT16) or SI:DI (FAT32) registers to hold the number of the first cluster of the first file. This is used by the MS-DOS v7 protocol.

SET_DSSI_DPT

If true, initialise DS:SI registers to point to the DPT. (Set equal to the interrupt 1Eh vector.) This may be used by the MS-DOS v6 and IBMDOS protocols.

PUSH_DPT

If true, initialise stack to hold a segmented (16:16) pointer to the interrupt 1Eh vector (always 0:78h) and then the DPT address (equal to the interrupt 1Eh vector). This may be used by the MS-DOS v6 and IBMDOS protocols, and is used by the MS-DOS v7 protocol.

DATASTART_HIDDEN

If true, modify the data start variable at `dword [ss:bp - 4]` to include the number of hidden sectors. (The hidden sectors are the partition's start offset in its unit.) This is used by the MS-DOS (v6/v7) and IBMDOS protocols.

SET_AXBX_DATASTART

If true, set the AX:BX register pair to the data start variable. If DATASTART_HIDDEN is also set, the registers will receive the value of the data start variable that includes the hidden sectors. This is used by the MS-DOS v6 and IBMDOS protocols.

SET_DSBP_BPB

If true, set up the DS register to equal SS. This makes DS:BP point to the boot sector with the (E)BPB. This is used by the EDR-DOS protocol.

LBA_SET_TYPE

If true, change the third byte of the boot sector to indicate the use of LBA access functions in the manner expected by the MS-DOS v7 load protocol. That means a 90h (nop instruction) is written if to use CHS access, a 0Eh is written if the FAT type is not FAT32 and to use LBA access, and a 0Ch is written if the FAT type is FAT32 and to use LBA access.

MESSAGE_TABLE

If true, include the message table used by the MS-DOS v7 load protocol.

SET_AXBX_ROOT_HIDDEN

If true, pass the sector number of the root directory start including hidden sectors in the AX:BX register pair. This is used by the RxDOS.0 and RxDOS.1 protocols.

NO_BPB

If true, do not load the boot sector with BPB. This is used by the CHAIN protocol.

SET_DSSI_PARTINFO

If true, load sector with partition table to address 00600h and point DS:SI and DS:BP to the active partition table entry within the partition table. This is used by the CHAIN protocol.

CMDLINE

If true, allow a command line to be specified after the filenames. This is used by the RxDOS.2, RxDOS.3, and IDOS protocols. As an extension it may be enabled for the EDR-DOS or FreeDOS protocols, too.

10.9.1.3 Specifying protocol partition

After the parameters, the debugger will try to parse a partition specification. Partition specifications are capitalisation-insensitive. A partition may be specified in the following ways:

FDA

Diskette-style (unpartitioned) file system on unit 00h

FDB

Diskette-style (unpartitioned) file system on unit 01h

HDA1

MBR-style (partitioned) file system on unit 80h, first primary partition

HDA2

MBR-style (partitioned) file system on unit 80h, second primary partition

HDA5

MBR-style (partitioned) file system on unit 80h, first logical partition

HDA6

MBR-style (partitioned) file system on unit 80h, second logical partition

HDA(partnumber)

MBR-style (partitioned) file system on unit 80h, with partition specified by partnumber.

HDB1

MBR-style (partitioned) file system on unit 81h, first primary partition

LDP

File system that the debugger loaded from

YDP

File system that the most recent Y command loaded from

SDP

Last used file system

U00.

Diskette-style (unpartitioned) file system on unit 00h

U80.1

MBR-style (partitioned) file system on unit 80h, first primary partition

U(unitnumber).(partnumber)

Unit specified by unitnumber with partition specified by partnumber. The specified partnumber may be specified as an expression in parentheses or as a literal number without parentheses. If it is an expression equal to zero in parentheses then that means unpartitioned.

LD(partnumber)

Unit that the debugger loaded from, with partition specified by partnumber.

YD(partnumber)

Unit that the most recent Y command loaded from, with partition specified by partnumber.

SD(partnumber)

Last used unit, with partition specified by partnumber.

If no partition can be parsed, SDP is assumed. Note that partition numbers are parsed as decimal numbers, except if the partition number is specified as an expression with parentheses, in which case the default expression base is used (hexadecimal).

10.9.1.4 Specifying protocol filenames

One or two pathnames may be specified to load, after the parameters or the partition specification. Both will have a default specified by the protocol. The default for the second name may be empty. If the second name is empty, no additional file is searched for. If two names are specified, they must be separated by one or more blanks.

Each pathname may include subdirectory names, indicated by trailing slashes. If a pathname ends in a slash, the default filename is searched in the directory indicated by the pathname. The second pathname may be specified as two slashes to indicate no second file. If the second name defaults to empty, a lone dot may be used to indicate no second file. If the second name defaults to not empty, a lone dot may be used to indicate searching the default name in the same directory as the first file. If either pathname is specified as only one slash, then the corresponding default name is searched for in the root directory.

If the second name is not specified at all, or is specified as a lone dot, the default additional filename is searched for in the same directory as the first file. If the second name is specified but does not start with a slash, then it is assumed to be a pathname relative to the directory of the first file. Otherwise, if the second name is specified and starts with a slash, the second name is searched in the directory of the specified pathname which is interpreted as being relative to the root directory.

The 32-byte directory entry of the first file is loaded to 00500h. The 32-byte directory entry of the second file is loaded to 00520h. These entries are used by the MS-DOS v6 and IBMDOS protocols. If no second file is searched for, the 32 bytes at 00520h are filled with zeroes.

The blank-padded FCB filenames of the two files are stored within the pseudo boot sector with (E)BPB that the loader sets up for the kernel. This supports kernels scanning the boot sector for informational filenames.

10.9.1.5 Specifying protocol command line

If the CMDLINE boolean parameter is enabled, then after the two pathnames specifying filenames a command line is parsed. The command line should be separated from the second filename specification with one or more blanks. This command line is passed to the loaded kernel as specified for the IDOS load protocol. (The FreeDOS kernel was extended to also use a command line passed this way.)

If the CMDLINE parameter is enabled but no command line content is specified, then an empty command line is passed. Note that this differs from passing no command line. To pass no command line, the CMDLINE parameter must be disabled.

As a special case, semicolons are allowed within the specified command line and do not indicate comments.

10.9.1.6 Boot load protocol compatibilities

The following notes are from IDOS boot.asm, in a comment titled ‘Notes about partial load compatibilities’:

10.9.1.6.1 FreeDOS

- Relocates to an address other than 27A00h (1FE0h:7C00h)

This is true of the IDebug BOOT command as well. However, the original FreeDOS loader relocation can be emulated by specifying a BPB=1FE0:7C00 parameter.

- A lot of options between _USE_PART_INFO, _QUERY_GEOMETRY, _CHS, _LBA, and/or _RPL need to be disabled to make the loader fit

This is not a problem to the IDebug BOOT command.

10.9.1.6.2 DR-DOS

- Must enable _MEMORY_CONTINUE to load off file systems with cluster sizes > 1 KiB (depending on file size) or >= 32 KiB (certainly)
- Will not load whole file if _MEMORY_CONTINUE is enabled and file data exceeds 29 KiB, without erroring out. The new option called _LOAD_CHECK_MAX_SIZE has been added to address this. However, it won't fit without disabling some other options such as _LBA, _CHS, _USE_PART_INFO, or _QUERY_GEOMETRY.

This is not a problem to the IDebug BOOT command. The full 29 KiB can be loaded regardless of sector size and cluster size.

- Disables DPT, data start, and directory entry to 00500h options by default so loading a faithful IBMDOS v4+ kernel will fail as msload or IDOS iniload need these options to operate

This is not a problem to the IDebug BOOT command. In fact the DRDOS protocol in IDebug exactly matches the IBMDOS protocol except that it defaults to a MAXPARA=-1 parameter. Therefore a small enough IBMBIO.COM file can be loaded with the DRDOS protocol even if it expects any of the bits and bobs provided by the IBMDOS protocol.

10.9.1.6.3 IBMDOS and MS-DOS 6

- Does not actually relocate DPT, just provide its address

Applicable to IDebug as well. The table can be relocated and modified by the user however, if need be. This should be done after running a BOOT PROTOCOL command so as to pass the original DPT address to the kernel.

- A lot of options between _USE_PART_INFO, _QUERY_GEOMETRY, _CHS, and/or _LBA need to be disabled to make the loader fit

This is not a problem to the IDebug BOOT command.

10.9.1.6.4 MS-DOS 7

- Does not actually relocate DPT, just provide its address

Applicable to IDebug as well. The table can be relocated and modified by the user however, if need be. This should be done after running a BOOT PROTOCOL command so as to pass the original DPT address to the kernel.

- Does not contain message table used by loader

The message table is provided by the IDebug BOOT command if enabled using the MESSAGE_TABLE parameter. This is default enabled for the MSDOS7 protocol.

10.9.1.7 Boot load protocol compatibilities additions

10.9.1.7.1 FreeDOS

The original FreeDOS loaders may misbehave trying to load a file that is larger than 128 KiB, when rounded up to a full cluster boundary. The lDebug loader is impacted by this problem less.

10.9.1.7.2 DR-DOS

The DR-DOS and the original Enhanced DR-DOS load (prior to lDOS single-file EDR-DOS load in 2024) are similar. They depend on the BIO file to be fully loaded. The BIO file is limited to 29 KiB for DR-DOS load. (For original EDR-DOS load the BIO file has limits similar to the kernel file for FreeDOS load.)

These loads do not use any directory entries nor file start clusters passed anywhere. Consequently, they have to scan the root directory for their DOS files, named IBMDOS .COM or DRDOS .SYS. That means these files and their filenames cannot be overridden by the debugger.

DR-DOS load presumably uses only the load unit passed in the DL register to find the file system that it wants to load its DOS file off. Original EDR-DOS load also used the hidden sectors in the BPB pointed to by DS:BP to identify the correct file system.

10.9.1.7.3 IBMDOS and MS-DOS 6

Although the initial loader originally introduced for MS-DOS v4.00 is called the "Non-Contiguous IBMBIO Loader (MSLOAD)" it does still depend on the first cluster of the BIO file being cluster 2, the first data cluster of the file system. This was fixed as of MS-DOS v5.00 however.

Original loaders expect the BIO file as the first entry in the root directory and the DOS file as the second entry. They also arrange for these entries to be loaded to linear 00500h and 00520h. The debugger will allow any file in the file system to be used and will load the directory entries to the expected locations, which may enable booting differently named or located files.

There is a problem with the DOS file loader: It may select a drive to load from using only the load unit, but read the start cluster from the DOS file directory entry and use the data start sector passed from the prior loader.

10.9.2 BOOT LIST command

```
BOOT LIST [unit|partition]
```

This command is used to list partitions on an MBR-style partitioned unit.

10.9.3 BOOT DIR command

```
BOOT DIR [partition] [pathname]
```

This command is used to list files within a directory of a FAT12, FAT16, or FAT32 file system. A partition specification may be included. A pathname may be included; if it refers to a directory then the contents are listed, otherwise the specified file is listed.

10.9.4 BOOT READ and BOOT WRITE commands

```
BOOT READ|WRITE [unit|partition] segment [[HIDDEN=sector] sector [count]]
```


These commands are used to read or write sectors from disks. After the command keyword, a partition specification may be listed. Then, a segment must follow. This specifies the buffer to use.

After the segment, an optional `HIDDEN=` keyword can be specified to specify a 32-bit sector number base. This is useful to implement 33-bit LBA access, but note it will overwrite the partition offset if a partition is specified. Instead of the `HIDDEN=` keyword, a `HIDDENADD=` keyword can be specified. It also reads a 32-bit sector number base. However, as opposed to `HIDDEN=`, `HIDDENADD=` will add to the hidden value of a specified partition, instead of replacing it. If a whole unit without a partition is specified then `HIDDEN=` and `HIDDENADD=` will result in the same offset being used.

After the segment and optional hidden keywords, a 32-bit sector number may be specified. It defaults to zero. After the sector number, a 16-bit count may be specified. It defaults to one.

Note that sectors are read or written one sector at a time. If the interrupt 13h function used returns an error 9 (boundary error), the debugger will attempt to use its auxiliary buffer to carry out the read or write, copying data as appropriate. The auxiliary buffer is aligned so as not to cross a 64 KiB boundary in memory. (Error 9 is usually returned if trying to access too many sectors at once or when diskette ISA DMA would cross a 64 KiB boundary.)

10.9.5 **BOOT QUIT command**

`BOOT QUIT`

This attempts to shut down the machine. A `dosemu`-specific callout will be attempted first, if `dosemu` is detected. Using `APM` will be attempted next, which works on `qemu`. If neither works, the debugger gives up.

10.10 **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.

10.11 **COUNT command - Count list length**

`count length` `COUNT [RANGE range|list]`

This command parses a range or list parameter. The resulting pattern's length is displayed in hexadecimal and decimal. The `COUNT` variable is also set to the length count number. If a range is given then the length has to fit into the segment.

10.12 **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. The range may be specified with a lines length (refer to section 8.4). The default length if none is specified defaults to the number of lines specified

in the variable `DEFAULTDLINES` if it is nonzero, or else the number of bytes specified in the variable `DEFAULTDLEN`.

If a lines length is used, that many lines are dumped. The count of lines does not include the header or trailer if they're used. The count of lines will be inaccurate if the symbolic build is used and the dump lists symbols that point into the dumped data. (The amount of data in this case will match what it should be to produce the requested count of lines if no symbols were listed.) The count of lines will be accurate if either the 40-column friendly mode is enabled or not. If enabled, roughly half as much data is dumped for a given amount of lines, as the 40-column mode dumps up to 8 bytes per line as opposed to the 80-column mode which dumps up to 16 bytes per line.

If the DCO option 4 is set, text with the high bit set (80h to FFh, the top half of the 8-bit encoding space) is displayed as-is in the text dump. If a `TOP` keyword is used before the range, then top half text is displayed as-is as well. Otherwise, it will be treated like nonprintable text, which means it is replaced by dots in the text dump.

The variable `DDTEXTAND` is used as a mask to modify the text code before display. It defaults to 0FFh. Setting this to 7Fh will mimic MS-DOS Debug's display of text in its data dump, masking off the high bit. In this case the `TOP` setting has no effect.

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 length is determined in the same way as for if a range without a length is specified. If `autorepeat` is used it behaves the same way as a `D` command without a range.

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`.

10.13 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 the T/TP/P silent buffer is used, or if RH mode is enabled. 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.

10.14 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 or device driver, 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 or device driver.

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: 2ACA
02B4 4D 0008 0018      384 B SD
02CD 4D 0000 0013      304 B
02E1 4D 02E2 00A8    2.6 KiB COMMAND
038A 4D 038B 00A8    2.6 KiB COMMAND
0433 4D 0434 2695    154 KiB LDEBUG
2AC9 5A 2ACA 7535    468 KiB DEBUGGEE
9FFF 4D 0008 2100    132 KiB SC
C100 4D 0008 0144    5.0 KiB SD
C245 4D 0000 0006      96 B
C24C 4D C24D 00A8    2.6 KiB COMMAND
C2F5 4D 0000 1D09    116 KiB
DFFF 4D 0008 1000   64.0 KiB SC
F000 4D F001 0019     400 B SEEKEXT
F01A 4D 0000 07F3   31.7 KiB
F80E 4D 0000 0090    2.2 KiB
```

```

F89F 4D 038B 001E      480 B COMMAND
F8BE 4D 02E2 0040      1024 B COMMAND
F8FF 5A C24D 0100      4.0 KiB 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. This is a number of up to 4 digits, which may have a fractional part, and a unit of B (Bytes) or KiB (Kilo binary Bytes).
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.

10.15 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.

10.16 D.A/D.D/D.B/D.L/D.T commands - Descriptor modification

These commands are only available in lDebugX (DPMI-enabled) builds. They can only be used in Protected Mode. RC is set to 800h when attempting to use any of these commands while not in Protected Mode.

Descriptor modification commands:

(only valid in Protected Mode)

```

Allocate      D.A
Deallocate    D.D selector
Set base      D.B selector base
Set limit     D.L selector limit
Set type      D.T selector type

```

10.16.1 D.A command - Allocate descriptor

Allocate D.A

Allocates an LDT descriptor from the DPMI host. Sets the variable DARESULT to the selector if successful, else FFFFh. Sets RC to 801h or a DPMI error code ($\geq 8000h$) on failure.

10.16.2 D.D command - Deallocate descriptor

Deallocate D.D selector

Deallocates an LDT descriptor. Sets RC to 802h or a DPMI error code ($\geq 8000h$) on failure.

10.16.3 D.B command - Set descriptor base

Set base D.B selector base

Sets the base of an LDT descriptor. A useful shorthand is to use a construct like 'LINEAR cs:0' to get the base of a descriptor referenced by another selector. Sets RC to 803h or a DPMI error code ($\geq 8000h$) on failure.

10.16.4 D.L command - Set descriptor limit

Set limit D.L selector limit

Sets the limit of an LDT descriptor. Limits beyond FFFFFh must be 4 KiB aligned (low 12 bits set). Sets RC to 804h or a DPMI error code ($\geq 8000h$) on failure.

10.16.5 D.T command - Set descriptor type

Set type D.T selector type

Sets the type of an LDT descriptor. 00FAh is a 16-bit code segment, 4000h is the D/B bit (Default size / Big), so 40FAh is a 32-bit code segment. 00F2h is a 16-bit data segment. 8000h is the G bit (Granularity); modifying it may change the limit. The DESCTYPE keyword can be used in an expression to read the current type of a descriptor, refer to section 9.7. Sets RC to 805h or a DPMI error code ($\geq 8000h$) on failure.

10.17 DT command - Dump text table

dump text table DT [T] [number]

Without a number parameter and without the T specifier, an ASCII table (codepoints 00h to 7Fh) is listed with short names for unprintable ASCII but the text itself for printable ASCII. The table lists the decimal and hexadecimal numbers. Without a number parameter but with the T specifier, a similar table depicting the top half of the 8-bit codepoint space is listed (values 80h to FFh).

With a parameter, each byte of the specified number is displayed in decimal, hexadecimal, and the short name or the text itself (quoted). If the T specifier is present, top half text (value $\geq 80h$) is displayed quoted, otherwise it instead displays as 'top'. The command will loop starting with the least-significant byte of the number, then continue with subsequent bytes until all remaining

bytes are equal to zero. All up to four bytes are listed on the same line.

Note that without the T specifier, no codepage dependent text is displayed. The T stands for 'top'.

10.18 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. If no address is specified then interactive enter mode starts at the last used address. This is behind the last byte written by a prior E command, or at the last byte displayed in interactive enter mode.

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
-
```

10.19 EXT command - Load and run an Extension for IDebug

```
run extension    EXT [partition/][extensionfile] [parameters]
```

The EXT command is used to run an Extension for IDebug (ELD) file.

This command and the infrastructure needed for it, including two buffers of usually several dozen KiB together, are only included when the debugger is built with the `_EXTENSIONS` option enabled. (This is now the default.)

The ELD must be in a special executable format defined by the debugger. The current ELD format starts with the magic bytes 'ELD1' at offset zero in the file. An ELD file typically has a filename extension `.ELD` or `.XLD` though this is not required. (If the `extname.eld` is installed it will check for either of the two known extensions, and if none is specified it will guess the same two extensions in order.)

To load ELDs in the application mode or device mode debugger, the DOS file system is used. That means DOS must be available for loading ELDs then. To load ELDs in the boot loaded mode debugger, the auxiliary buffer must be available and int 13h is used to access a FAT file system.

Like the Y command (refer to section 10.58.1), the EXT command pathname can be specified with one of the debugger configuration path keywords. Also, if an EXT command doesn't find a file specified without a configuration path keyword, the debugger will retry the file open with the `::scripts::` path prepended.

Unlike Script for IDebug files opened with the Y command, Extensions for IDebug may be run with parameters specified after the ELD filename. The ELD receives the parameters as free form string data which it is free to interpret as it wishes. A semicolon may or may not be interpreted as a comment indicator by an ELD. The ELD can expect that within 256 bytes a Carriage Return occurs.

Some ELDs can install themselves as resident Extensions to the debugger, hooking into the command dispatch to run their own command rather than the debugger's default. All resident ELDs provide a command hook, at least for their respective uninstall commands. Some resident ELDs may hook into other parts of the debugger as well.

10.19.1 Current ELDs

The ELDs are described in detail in section 15. The following ELDs are provided currently:

LDMEM

Displays information on memory use of the debugger. Can be installed residently with `INSTALL` parameter to provide the `LDMEM` command, and uninstalled with an `LDMEM UNINSTALL` command.

HISTORY

Lists the command history of the debugger, or clears it. Can be installed residently with `INSTALL` parameter to provide the `HISTORY` command, and uninstalled with a `HISTORY UNINSTALL` command.

DI

Re-creation of the DI and DIL commands of the debugger. This allows to use the DI commands when the debugger is built with the `_INT=0` build option. Can be installed residently with `INSTALL` parameter to provide the DI command, and uninstalled with a `DI UNINSTALL` command.

DM

Re-creation of the DM command of the debugger. This allows to use the DM command when the debugger is built with the `_MCB=0` build option. Can be installed residently with `INSTALL` parameter to provide the DM command, and uninstalled with a `DM UNINSTALL` command.

RN

Re-creation of the RN command of the debugger. This allows to use the RN command when the debugger is built with the `_RN=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the RN command, and uninstalled with an `RN UNINSTALL` command.

RM

Re-creation of the RM command of the debugger. This allows to use the RM command when the debugger is built with the `_RM=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the RM command, and uninstalled with an `RM UNINSTALL` command.

X

Re-creation of the X commands of the debugger. This allows to use the X commands when the debugger is built with the `_EMS=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the X commands, and uninstalled with an `X UNINSTALL` command.

DX

Re-creation of the DX command of the debugger. This allows to use the DX command when the debugger is built with the `_DX=0` build option (which is the default as of release 9). Can be installed residently with `INSTALL` parameter to provide the DX command, and uninstalled with an `DX UNINSTALL` command. This ELD requires a 386+ machine.

INSTNOUN

Displays information on install flags of the debugger. These are the nouns accepted by the `INSTALL` and `UNINSTALL` commands. Can be installed residently with `INSTALL` parameter to provide the `INSTNOUN` command, and uninstalled with an `INSTNOUN UNINSTALL` command.

RECLAIM

Transient utility to reclaim unused space in the ELD code buffer and the ELD data blocks buffer. This is no longer needed because the debugger now includes the implementation of this tool and automatically reclaims memory before loading an ELD.

ELDCOMP

A tool to compare an ELD with its XLD counterpart. XLD is the filename extension typically used to hold a build of an ELD with some linker optimisations. ELDCOMP allows to compare the two, helping to identify and locate relocation errors in the ELD to be tested.

AFORMAT

Once installed, this ELD hooks into the assembler. After a line is submitted to the assembler, the AFORMAT ELD will dump in hexadecimal the bytes written by the assembler.

AMISMSG

This ELD hooks into the debugger's AMIS interface. It provides the AMIS functions 40h and 41h to send messages to the debugger terminal. A message may consist of up to 383 bytes of text. The message is displayed either on the next command being read in the cmd3 command input loop using a command injection handler, or when the command 'AMISMSG DISPLAY' is run.

AMOUNT

This ELD once installed provides the ELDRAMOUNT variable. This variable can be read to obtain the amount of installed ELDs.

BASES

A converter for different numeric bases. Can accept a numeric parameter to be evaluated by the expression evaluator. Using a BASE= specifier or the name of a base (HEXADECIMAL, DECIMAL, BINARY, OCTAL) the input number may be specified as a literal, which accepts unsigned numbers of up to 64 bits. Output is in the four known bases, formatted to resemble the expression evaluator's literal input format. Output can be in one arbitrary base using a trailing OUTPUT keyword, followed by BASE=, GROUP=, and WIDTH= specifiers. The BASES ELD can be installed residently using an INSTALL parameter to provide the resident BASES command.

CO

Installs the COPYOUTPUT commands. Once a file is specified with 'COPYOUTPUT NAME filename' the debugger opens the file to append to it. While not InDOS all output to the debugger terminal is written to the opened file.

CONFIG

Allows to show or set the debugger config paths.

DTADISP

Displays the current DOS Disk Transfer Address.

IFEXT

Allows to run another command conditionally, using a command of the form IF [NOT] EXT "extension name" THEN command. May be used as a transient ELD or installed residently using an INSTALL command to provide the IF EXT commands.

KDISPLAY

Displays the current K/N command buffers' content.

LIST

Displays the description lines for ELD files that are specified with a single pathname pattern. The pattern may contain wildcards in the last component. After the pathname, several keywords may be specified: `VERBOSE` to display technical details, `HELP` to display the help, and `SFN` to force use of the DOS SFN find interface instead of trying the DOS LFN find interface.

PRINTF

Allows to print formatted output. Can be installed residently or used as a transient tool.

SET

Allows to access the environment block to read or write variables. Can install a resident SET command using an `INSTALL` parameter, or run transiently using a `RUN` parameter.

USEPARAT

Installs an output hook into the debugger to display an underscore line after disassembling near or far jumps or near, far, or interrupt return instructions.

VARIABLE

Installs a command preprocessor hook to expand '`%VARIABLE%`' specifications in commands.

WITHHDR

Installs a prefix command called `WITH`. This can be used as `WITH HEADER` or `WITH TRAILER` to temporarily set DCO flags to enable D command headers or trailers. Command injection is used to reset the flags afterwards.

AMISCMD

This ELD hooks into the debugger's AMIS interface. It provides the AMIS function 43h to inject commands into the debugger.

AMISOTH

This ELD hooks into the debugger's AMIS interface. It provides the AMIS function 42h to export the debugger's link info as an "other link".

AMITSRS

Port of Ralf Brown's AMIS TSR lister.

BOOTDIR

List directory entries in bootloaded mode.

DBITMAP

Dump 8-bit-wide graphics from memory.

DOSCD

Change DOS current directory or drive.

DOSDIR

List directory entries using DOS.

DOSDRIVE

Get or set a DOS drive.

DOSPWD

Display DOS current directory.

EXTNAME

Installs residently to guess EXT and Y command filename extensions.

INJECT

Injects commands into other debugger instance using the other's AMIS function 43h. (This function is provided by the AMISCMD ELD.)

INSTNOTH

INSTNOUN variant that operates on another debugger instance. This requires the other instance to have installed the AMISOTH ELD and its AMIS handler to provide the AMIS function 42h.

LDMEMOTH

LDMEM variant that operates on another debugger instance. This requires the other instance to have installed the AMISOTH ELD and its AMIS handler to provide the AMIS function 42h.

LINFO

Installs residently to display status of program-loading L command.

PATH

Provides path search and filename extension guessing for the K and N commands.

EXTLIB

Library of ELDs to be used instead of single files.

EXTPAK

Compressed library of ELDs.

QUIT

Quits the machine. Can be installed residently.

DOSSEEK

Get or set the DOS 32-bit seek of a process handle.

ALIAS

Define simple aliases that are replaced in a command preprocess handler.

DPB

Display a DOS drive's DPB (MS-DOS v4 to v6 layout, optionally with FreeDOS, MS-DOS v7.10, or EDR-DOS FAT32 extensions).

RDumpIdx

Dump text bytes pointed to by DS:SI and ES:DI in R register dump.

RDumpStr

Dump text pointed to by DS:DX in R register dump.

CHECKSUM

Calculate checksum over a memory range, as a transient command or installed residently.

HINT

Display TracList listing offset hints for all installed ELDs, writing to the terminal of another debugger instance (using its AMISMSG service). Can be used as a transient command or installed residently.

HINTOTH

Display TracList listing offset hints for all installed ELDs of the other link debugger instance (using its AMISOTH service). Can be used as a transient command or installed residently.

CHSTOOL

Utilities to work with int 13h unit partitions and geometry. Can be used as a transient command or installed residently.

S

Replaces the S command (section 10.47) with additional support for WILD and CAPS keywords.

10.20 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.

10.21 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. The build option `_NUM_G_BP` specifies how many G breakpoints are supported. 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.

10.22 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.

10.23 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
-
```

10.24 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.

10.25 IF command - Control flow conditional

```
if numeric      IF [NOT] (cond) THEN cmd
if script file  IF [NOT] EXISTS Y file [:label] THEN cmd
if variable     IF [NOT] EXISTS R variablename 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 10.22).

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 10.58). 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.

The third form checks for a variable name being recognised like for the R variable access command. If the variable name is recognised the condition is considered true. Otherwise, the candidate name is skipped by scanning for the first blank or comma, except for parenthetical index expressions which are parsed as expressions. The condition is considered false if no variable is recognised.

10.26 INSTALL command - Install optional features

This command can be used to enable certain optional features. The parameters are a list of comma-separated keywords. First the entire list will be parsed. Upon successful parsing of all keywords the command will then start to handle the keywords.

The first keyword to the INSTALL command may be the TOGGLE keyword. In this case the subsequently specified keywords are toggled rather than enabled.

Save for the 'AREAS' keyword, these features can be accessed by using the corresponding DCO flags as well. The allowed keywords are:

INT2F

DPMIHOOK

(lDebugX-only) Enable installing debugger's interrupt 2Fh hook to intercept the DPMI entryptoint function call. The interrupt hook will actually occur upon running any debuggee code in Real/Virtual 86 Mode. If the debugger is unable to hook the DPMI entryptoint then a message is displayed and the DCO4 flag is cleared. This is expected on MSWindows 4 and old dosemu versions. This hook is enabled by default. This keyword controls DCO4 flag 0002h.

FAULTS

INTFAULTS

Install debugger's interrupt 0Dh and 0Ch hook. These interrupts may be called by the machine in Real 86 Mode, or by a VMM like dosemu2 in Virtual 86 Mode, to indicate

faults occurred. Interrupt 0Dh indicates a General Protection Fault, while interrupt 0Ch indicates a Stack Fault. (If an address faults with the SS segment it is considered a Stack Fault.) Both of these interrupt handlers are usually called for IRQs as well, however. The debugger's handlers will check whether the corresponding IRQ is being serviced. If it is, the call is chained to the debugger's downlink. Only if the IRQ is not being serviced, the debugger will handle the call as a fault. (This heuristic is not perfect, but it works most of the time.) This keyword controls DCO4 flag 0010h.

INT08

INT8

TIMER

Install debugger's interrupt 8 hook for the timer tick IRQ. This enables the Interrupt 8 Control pressed detection depending on the INT8CTRL variable (refer to section 12.27) as well as the double Control-C via serial I/O detection. This keyword controls DCO4 flag 0004h.

INT2D

AMIS

Install debugger's AMIS interface on interrupt 2Dh. A free multiplex number must be available and the existing interrupt vector must be valid for this to succeed. This keyword controls DCO4 flag 0008h.

AREAS

(IDDebugX-/ICDebugX-only) Install this debugger's exception areas into another debugger. The other debugger must have its AMIS interface installed at the point in time that this command is run. This keyword does not correspond to any DCO flags.

SERIAL

Install serial I/O for the debugger interface. Configuration must be ready, refer to section 12.11. This keyword controls DCO flag 4000h.

INDOS

Enter force InDOS mode. The debugger will avoid calling DOS in this mode, helping in debugging DOS or the interrupt 21h or 2Fh handlers. This keyword controls DCO flag 0008h.

GETINPUT

Enter the use DOS getinput mode. When DOS is available and used for I/O, this mode enables use of the debugger's `getinput` function rather than using the DOS interrupt 21h service 0Ah line editor to read from standard input. The debugger's line editor includes proper line editing, allows overlong input with a horizontally scrolling view of the buffer, and enables history recall. This keyword controls DCO flag 0800h.

RHIGHLIGHT

Enable R command register change highlighting. This keyword controls DCO3 flag 04_0000h.

AUTOREPEAT

Enable autorepeat while reading from a terminal. This feature is enabled by default. This keyword controls DCO3 flag 1000_0000h (in reverse).

BIOSOUTPUT

Prefer to output to video ROM-BIOS terminal using interrupt 10h services instead of to the DOS interrupt 21h. This keyword controls DCO6 flag 0200h.

FLATBINARY

FSWITCH

Enable flat binary read mode, in which MZ .EXE files and .HEX files are read and written as if they were flat binaries. This feature is also controlled by the /F+ or /F- switches. This keyword controls DCO6 flag 0400h.

BIGSTACK

ESWITCH

Enable .BIG style stack mode, in which flat binaries are loaded with the stack set up in a separate segment. This allows to execute .BIG style flat format executables like used in the build process of the debugger itself. This feature is also controlled by the /E+ or /E- switches. This keyword controls DCO6 flag 0800h.

RH

Enable RH mode. In RH (Register dump History) mode, a number of the last R, RE, T, TP, P, or G outputs are buffered in the auxiliary buffer. (The auxiliary buffer is about 8 KiB large by default.) The RH command can then be used to display all or some of the steps of the buffered contents. Other commands that use the auxiliary buffer, like RN and RM, cannot run when this mode is enabled. Enabling RH mode will clear the RH/silent buffer. This keyword controls DCO6 flag 10_0000h.

DEBUG

(ICDebug-only) Enable debuggable mode. The conditionally debuggable debugger defaults to start up in debuggable mode. This feature is also controlled by the /D+ or /D- switches. This keyword controls DCO6 flag 0100h.

PAGING

Enable paging. This feature is enabled by default. This keyword controls the DCO1 flag 10h (in reverse).

PAGINGRC

Enable paging when running RC buffer commands. This keyword controls the DCO3 flag 1000h.

PAGINGY

PAGINGSRIPT

Enable paging when running Script for IDebug commands. This keyword controls the

DCO3 flag 2000h.

PAGINGRE

Enable paging when running RE buffer commands. This keyword controls the DCO3 flag 4000h.

RX

REGS386

Enable 32-bit and 386 registers display. This keyword controls the DCO1 flag 1. This flag is also toggled by the RX command, section 10.41.

TM

TRACEINTS

Enables tracing into software interrupt handlers. This keyword controls the DCO1 flag 2. This flag is also accessed by the TM command, section 10.50.

HOUDINI

Enables breaking on houdini breakpoints in Extensions for IDebug. This only takes effect in IDDebug, or in ICDebug with debuggable mode enabled. This keyword controls the DCO7 flag 100h.

DEBUGOPTLINK

Enables debugging output of missing optional links in Extension for IDebug linker. This keyword controls the DCO7 flag 2.

QUIETINSTALL

Enables suppressing Extension for IDebug installation messages. This keyword controls the DCO7 flag 4.

QUICKRUN

Enables quickly running Extensions for IDebug. This keyword controls the DCO7 flag 8. Only quit.eld is affected by this yet.

10.27 L command - Load Program

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

10.28 L command - Load Sectors

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

Loads sectors from a local DOS drive. The count specifies how many sectors to load. The sector number specifies where on the drive to load from. The drive specifies the drive to access, and is specified either as an expression or a drive letter with trailing colon (section 8.13). The address is a segmented address indicating where to write the sector data.

The reverse is the Write Sectors command, section 10.56.

10.29 M command - Move memory

`move` `M range address`

This command copies data from one memory range to another range. The source range is specified as the first parameter. The destination is specified as the second parameter, which only accepts an address. The length of the destination is equal to the length of the source.

If the source and destination overlap, the data movement is insured to be done in the direction which will end up with the original source data found intact at the destination. If the source and destination do not overlap then the movement may be done forwards or backwards.

10.30 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.

Note that all machine mode commands that parse a numeric expression (not 'M', 'M ?', nor 'M NC') will actually parse the expression twice due to the internal dispatching between the machine mode commands and the move memory command. If the expression has side-effects then these side-effects will also occur twice. (An example of a side effect is reading the LFSR variable, which will step the LFSR.)

10.31 N command - Set program Name

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

This command sets up the filename and parameters to use when setting up a new process using the L (Load program) command. If the filename ends in .COM or .EXE it will be loaded as a DOS program using the interrupt 21h service 4B01h. If the filename ends in .HEX it will be interpreted by the debugger to load the contained data as a binary image. Otherwise the file is loaded as a flat binary by the debugger itself. In any case, the PSP of the process created by the L command will receive the command line tail, which starts after the filename.

Unlike Microsoft's Debug the executable filename is not included in the command line tail, and

an existing process won't be modified by the N command. It only sets the filename and tail for L to use.

10.32 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.

10.33 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.

10.34 Q command - Quit

quit Q

This command attempts to quit the debugger. It may fail if the currently attached process (if any) does not return into the debugger's Parent Return Address upon running an interrupt 21h function 4C00h in the debuggee context. It may also fail if any of the hooked interrupts cannot be restored. In this case, setting the corresponding DCO4 flags can force unhooking upon a retry.

For quitting a device driver mode debugger, the QC and/or QD flags to the quit command must be used. Refer to section 5.3 for a description of them.

If the debugger was attached to a DOS process the quit command will try to terminate the process. If the quit command succeeds, the debugger will return to its own parent process.

If not attached to a DOS process the quit command acts differently. This is always true of the bootloaded debugger, is true of the device driver mode debugger by default (absent any

ATTACH commands), and is true of the application mode debugger after a successful TSR command (absent subsequent ATTACH commands). In this case, a quit command that succeeds will continue to run the current debuggee code in the exact state it was left in last.

The bootloaded debugger offers the BOOT QUIT command which will try to quit the currently running (virtual) machine rather than quitting the debugger. It is described in section 10.9.5.

The quit Extension for IDebug (section 15.44) acts like the BOOT QUIT command but can be loaded as an ELD even when in a DOS mode.

If the QB flag to the quit command is used, then the debugger will run a breakpoint in its quit handler, if the quit command succeeds. This breakpoint runs after the debugger has uninstalled its interrupt handlers. Refer to section 10.36.

10.35 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 or bootloaded).

10.36 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.

10.37 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. If the instruction is a conditional jump then the R command disassembly will include a trailing notice that reads 'jumping' or 'not jumping' depending on the current status.

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
-
```

10.37.1 RE command - Run register dump Extended

Run R extended RE

The RE command runs the RE buffer commands. Refer to section 18.7. The RE buffer has the highest priority among all buffered commands. (The RC buffer and Y command Script for IDebug files have a lower priority than the RE buffer.)

RE buffer commands are displayed with a prompt consisting of a percent sign % or, for DDebug or for CDebug while in debuggable mode, a tilde followed by a percent sign ~%.

10.37.2 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. This command can overflow the RE buffer, in which case the command aborts with an error. In this case the command has no effect on the RE buffer contents.

RE.REPLACE replaces the RE buffer with the following commands. This command generally cannot overflow the RE buffer.

The RE buffer usage is described in the ?RE help page (section 18.7).

10.37.3 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. Further, the RC buffer contents have the lowest priority among all buffered commands. (The RE buffer and Y command Script for IDebug files have a higher priority than the RC buffer.) Upon initialisation of the debugger the RC buffer is filled.

In case the debugger is loaded as a DOS application or DOS device driver, the RC buffer first gets the configuration command. Then the debugger's init appends the content of the /C switch (if any). If an /IN switch is specified, the RC buffer is cleared.

In case the debugger is bootloaded, the RC buffer receives the contents of the kernel command line (if any) or the default kernel command line contents.

If the RC buffer is not empty, the equivalent to an RC command is run on startup of the debugger. (This running happens after the initial N and L commands.)

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

10.37.4 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. Like RE.APPEND this will cause an error if the buffer overflows.

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

10.38 RH command - Display Register dump History steps

regdump history RH [IN value, value, ...|value]

The RH command displays steps from the RH/silent buffer while RH mode is enabled. RH mode can be enabled using the INSTALL RH command. (Refer to section 10.26.) The parameter to the RH command can be the keyword IN, followed by one or more comma-separated match ranges, or by a single number, or no parameter at all.

The no parameter form displays all steps still saved in the RH buffer.

The one parameter form displays a single step from the RH buffer. The number 0 refers to the most-recent saved step. The number 1 refers to the second most-recent saved step. And so on.

The RH IN form accepts one or more match ranges. A match range can be:

- A single number. (Must be below 1_0000h.)
- The keyword FROM followed by a number followed by the keyword TO followed by a

number. (The second number must be above-or-equal the first number. Both must be below 1_0000h.)

- The keyword FROM followed by a number followed by the keyword LENGTH followed by a number. (The length number plus the first number must be below-or-equal 1_0000h. A zero-length is valid, and will produce no output.)

The RH IN command will treat each match range by displaying the corresponding steps from the RH buffer, but in chronological order. For instance, the following two commands are equivalent:

```
rh in from 2 length 4
```

```
rh in 5,4,3,2
```

When parameters specify a number that is above the oldest still saved step in the RH buffer, the behaviour is not certain and may yet change. (For now, each older step will display nothing. This still may be subject to change.)

The RH command defaults to page its output, if paging is enabled. Paging can be disabled for the RH command using the silent buffer output paging control. (This will, of course, also affect the silent buffer output.) The command `r dco3 = dco3 clr 200` or `100` will instruct the RH command to not page its output.

As an exception, if the RH buffer is empty then any valid RH command will produce no output at all.

If RH mode is not enabled then the RH command may display stale or corrupted output, except when run directly after a silent-buffered T/TP/P command. In the latter case, the RH command will operate on the contents stored by the last run command.

10.39 RM command - Display MMX Registers

```
MMX register      RM [BYTES|WORDS|DWORDS|QWORDS]
```

This command dumps all 8 MMX registers. It is only available if MMX is supported by the machine. The optional size keyword specifies an item size, which defaults to BYTES. The BYTES size will match memory order of the byte values, displaying the least significant byte's value first. A size keyword of WORD will display the least significant word first, and so on.

MMX support is redetected when lDebugX enters or leaves Protected Mode. (dosemu2 may support MMX in its Protected Mode while not supporting it in 86 Mode.)

This command is no longer included in the default build as of release 7. To use it, either enable the build time `_RM` define or run `'ext rm.eld install'` to install it as an Extension for lDebug.

10.40 RN command - Display FPU Registers

```
FPU register      RN
```

This command is no longer included in the default build as of release 7. To use it, either enable the build time `_RN` define or run `'ext rn.eld install'` to install it as an Extension for lDebug.

10.41 RX command - Toggle 386 Register Extensions display

```
toggle 386 regs RX
```

This command toggles the DCO flag 0001h. The same flag can be set using the command `INSTALL RX` or cleared using the command `UNINSTALL RX`. This command is not valid on a non-386 machine.

10.42 RV command - Show sundry variables

This command shows the first 16 user-defined variables (refer to section 12.16), 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). IDebugX 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 (IDebugX only) Protected Mode with either a 16-bit CS or a 32-bit CS.

10.43 RVV command - Show nonzero user-defined variables

This command shows all user-defined variables (refer to section 12.16) 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.

10.44 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
- Code2 segment (only if `_DUALCODE` build)
- Data segment
- Entry segment (same as data segment but with a code selector in PM)
- Message segment
- Auxbuff segment
- History segment

A more detailed list of the debugger's segments, including their sizes, can be obtained using the `ldmem` Extension for IDebug (section 15.1).

10.45 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 IDebugX)

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 IDebugX)

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.

10.46 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.

10.47 S command - Search memory

```
search          S range [REVERSE] [SILENT number] [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.

If the SILENT keyword is specified, a silent number must be parsed before the list or range parameter. The number specifies how many result lines are displayed at most. It is a 32-bit unsigned number that may take any value in the range, including zero. If the number is zero, only the amount of matches is displayed.

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 IDebugX, 16-bit otherwise. Unused SRO variables are zeroed out by a successful search. The COUNT variable is set to the length of the search string.

The display of search results is as follows:

- First, the result's segmented address.
- Then, the displacement of the following dump. This is displayed with a leading plus sign and some amount of hexadecimal digits (2, 4, 6, or 8 digits). The number is the length of the search pattern (which is also written to the COUNT variable).
- Then, a hexadecimal dump of the up to 16 bytes that follow the search string match at this point.
- Finally, the ASCII character dump of these up to 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.

10.48 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.

10.49 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 (section 10.50), interrupt instructions are also traced (into the interrupt handler) or proceeded past.

10.49.1 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.

10.50 TM command - Show or set Trace Mode

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

This instruction accesses the DCO flag 2. If run without an expression then the current status is displayed. Otherwise tracing into interrupts (for the T and TP commands) is enabled (nonzero expression) or disabled (zero expression).

10.51 TSR command - Enter TSR mode (Detach from process)

```
enter TSR mode     TSR
```

This command tries to find the PSP to which the debugger is attached. It starts the search at the current PSP and walks up the parent processes until finding the debugger. If a self-owned process is encountered the search is aborted.

Once found, the attached PSP is patched with the PRA and parent process noted down for the debugger itself. That is, the debugger's would-be parent is made the parent of the attached process. The debugger's fields for original PRA and parent are cleared. Thus the TSR command, if it succeeds, switches the debugger into resident mode.

The reverse operation is performed by the ATTACH command (see section 10.6). The TSR command can be used right away in application mode, or can be used in device driver mode after the debugger has been attached to a process using the ATTACH command. The default state of the device driver mode debugger is resident mode.

The TSR command and the ATTACH command are not usable in bootloaded mode.

10.52 U command - Disassemble

```
unassemble         U [range]
```

Given a range, the address of which defaults to CS, this command disassembles instructions from memory. The range may be specified with a lines length (refer to section 8.4). The default length if none is specified defaults to the number of lines specified in the variable DEFAULTULINES if it is nonzero, or else the number of bytes specified in the variable DEFAULTULEN.

If a lines length is used, that many lines are disassembled. However, if a single instruction does not fit within one line due to a too long string of machine code, then the lines used for this instruction will count as one line as concerns the lines length. If an address length is specified, all instructions that are contained within or start within the specified range are disassembled.

If no range is specified, the U command continues disassembling at "u_addr" (AUS:AUO), which is updated by each U command to point after the last disassembled byte. The R command sets "u_addr" to equal the current CS:(E)IP, including if the register dump is called by a run command. The default length is determined in the same way as for if a range without a length is specified. If autorepeat is used it behaves the same way as a U command without a range.

10.53 UNINSTALL command - Uninstall optional features

This command can be used to disable certain optional features. The parameters are a list of comma-separated keywords. First the entire list will be parsed. Upon successful parsing of all keywords the command will then start to handle the keywords.

The available keywords are documented for the INSTALL command, refer to section 10.26.

10.54 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.

10.55 W command - Write Program

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

10.56 W command - Write Sectors

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

Writes sectors to a local DOS drive. The count specifies how many sectors to write. The sector number specifies where on the drive to write to. The drive specifies the drive to access, and is specified either as an expression or a drive letter with trailing colon (section 8.13). The address is a segmented address indicating where to read the sector data.

The reverse is the Load Sectors command, section 10.28.

Note that on MS-DOS v7 this command will lock and unlock the specified drive to insure it is locked when writing. As there is no way to query the lock status, this will unconditionally unlock the drive after writing.

Warning: You should know what you are doing if you use this command. Misuse may corrupt any data stored on this drive.

10.57 X commands - Expanded Memory (EMS) commands

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

These commands are no longer included in the default build as of release 7. To use them, either enable the build time `_EMS` define or run `'ext x.eld install'` to install them as an Extension for IDebug.

10.58 Y command - Run script file

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

The Y command runs a Script for IDebug file.

10.58.1 Y command pathnames

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

- If running as an application or device driver, the script name is a regular pathname. It may be quoted with doublequotes if the pathname includes blanks, semicolons, or commas. 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 18.11 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.

In both cases, commas are parsed as separators that end a pathname, except if they occur within doublequotes. The same is true of semicolons. Semicolons may be parsed as end-of-line markers as well.

10.58.1.1 Y command configuration pathes

A pathname may start with one of two special keywords to use one of the debugger configuration pathes:

```
::config::
```

Access the debugger configuration directory

```
::scripts::
```

Access the debugger scripts directory

The pathname should not include a path separator (backslash or forward slash) directly after the keyword.

The default path for the debugger configuration directory is detected as follows:

1. If the debugger is bootloaded, the configuration directory is set to `ldp/`.
2. If the environment variable `LDEBUGCONFIG` is set, read it.
3. Else, if the path to the debugger's executable can be determined, use that.
4. Else, the current directory on the current drive is used.

The default path for the debugger scripts directory is detected as follows:

1. If the debugger is bootloaded, the scripts directory is set to `ldp/`.
2. If the environment variable `LDEBUGSCRIPTS` is set, read it.
3. Else, if the path to the debugger's executable can be determined, use that.
4. Else, the current directory on the current drive is used.

The configuration pathes can be shown or modified using the 'config' Extension for lDebug.

10.58.1.2 Y command default scripts path

If a Y command filename is not found, and no config path keyword was used, the debugger will retry the file open with the `::scripts::` path prepended to the specified filename. If this is not desired, an explicit `::empty::` path keyword may be used.

10.58.2 Y command labels

A label may be specified after a pathname 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.

10.58.3 Y command InDOS interaction

Opening a script file as DOS application or device driver 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.)

10.59 Z commands - Symbolic debugging support

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

10.59.1 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.

10.59.2 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.

10.59.3 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

10.59.4 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.

10.59.5 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.

10.59.6 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.

10.59.7 Z LIST - List symbols

10.59.8 Z MATCH - Match symbols

10.59.9 Z RELOC - Relocate symbols

Section 11: Assembler Reference

The assembler is accessed by running the A command (section 10.5). The prompt for the assembler consists of a segmented address, with a 16-bit segment/selector and a 16-bit or 32-bit offset. Entering an empty line (whitespace only) or a single dot to the assembler exits back to the debugger prompt. Trailing comments may be added using semicolons.

11.1 Assembler comparison to MSDebug

Differences from MSDebug's assembler include:

- 386 level 32-bit operands and addresses are supported. The call, jmp, push, and pop instructions with memory operands will assume a default WORD or DWORD size depending on the D bit of the current CS being assembled into
- Immediate operand and displacement sizes (BYTE, WORD, DWORD) can be specified to force longer encodings
- Disassembly defaults to NASM address syntax, lowercase output (except numeric digits), and a blank after any comma
- Indentation of the first operand can be disabled, though it defaults to enabled
- Some NASM style instruction names like `retn`, `int3`, and `xlatb` are used in the disassembly, and may be used with the assembler
- Segment overrides may be included in address operands rather than in a label-like form (on a line of their own or as a prefix to an instruction, with a trailing colon)
- Segment overrides do not require a colon if they are placed before string instructions
- Segment overrides may be placed in a SEG pseudo-instruction
- The disassembler always shows a size for push or pop with a memory operand, although the assembler can assume a default size

11.2 Assembler comparison to NASM

The assembler generally strives to accept assembly input matching the syntax of NASM (the Netwide Assembler). Key differences:

- Labels are not supported
- Numeric input defaults to hexadecimal (while expressions in parentheses can switch base using a '#' prefix)

- Character constants are not supported directly, requiring an expression in parentheses which accepts string literals
- SIB indexing must be done using exact numbers, multipliers like 9 or 5 are not automatically parsed into valid SIB bytes
- The NOSPLIT keyword is not supported, and the assembler never splits indexing expressions
- The MODRM keyword is added, which allows selecting non-default instruction encodings (most of which are size pessimisations)
- LOOP instructions can be suffixed with a size letter 'W' or 'D' whereas NASM requires a second operand that reads 'CX' or 'ECX'. The NASM form is supported too, but the disassembler defaults to the suffix form.
- A STRICT keyword is not accepted, instead size and distance keywords can be used to force certain forms
- CALL FAR and JMP FAR will allow a single immediate number operand, assembling to an imm:imm branch instruction with the segment equal to the segment currently being assembled into
- Size and distance keywords can be followed by a 'PTR' keyword, which has no effect

11.3 Assembler roundtrip

The disassembler is supposed to generate output which is valid input to the assembler. Ideally, any disassembly would make it through a "roundtrip", that is assembling the disassembly output should result in exactly the same machine code bytes. This is not always true however.

For instance, the SIB long form encoding of a disp32-only memory address is disassembled just as the non-SIB form, and the assembler currently lacks a way to force this encoding. Invalid prefixes may also not roundtrip successfully. The order of multiple prefixes may also change, which can be significant in some cases (eg the dispatchers for the disassembler's repeated string op simulation). Additionally, the disassembler may display annotations like '(unused)' and '[needs 386]'. These must be stripped from the instruction to enter it into the assembler. The same is true when running an R command (register dump) and its disassembly shows a memory content annotation or the 'jumping' or 'not jumping' notice.

11.4 Disassembly fields

The disassembly consists of several fields:

1. The segment which is being read, 4 hexadecimal digits
2. The offset of the first machine code byte in this line, 4 or 8 hexadecimal digits
3. The machine code dump, an even number of hexadecimal digits
4. The instruction mnemonic
5. 0 to 3 instruction operands, depending on the instruction
6. The annotation, optional

11.5 Assembly fields

The assembly input consists of:

1. The segment:offset that the next instruction will be written to, this forms the prompt for the debugger's line input. Offset may be 4 or 8 hexadecimal digits.
2. The instruction mnemonic
3. 0 to 3 instruction operands, depending on the instruction
4. A comment indicated by a semicolon, optional

The assembler may emit an annotation after it accepts a line. This will be written to the next line, and can read like '[needs 386]' or '[needs math coprocessor]' depending on the current machine type (section 10.30) and the needed machine type for the instruction.

Additionally, if the AFORMAT Extension for IDebug (section 15.12) is installed and enabled, each accepted line to the assembler will reply with a machine code dump in a subsequent line. This dump looks similar to the disassembler's dump. It will correspond exactly to what was written by the assembler. (It checks which address was used for the last prompt and which address is used for the next prompt. The ORG directive is handled as a special case.)

11.6 Assembly instruction reference

It is recommended to keep an instruction reference on hand. The old NASM instruction reference does nicely for all 386-level instructions, though it lacks descriptions of system structures like descriptors, the TSS, and the FSAVE format. It is hosted on the web at <https://pushbx.org/ecm/doc/insref.htm> with its sources available in the repo at <https://hg.pushbx.org/ecm/insref/>

11.6.1 Assembler instruction mnemonics

A mnemonic is a keyword that maps to a certain instruction. Mnemonics are generally matched cap-insensitively. Some mnemonics allow different forms with optional or required size suffix letters. If present, such a letter can be either a 'W' or a 'D' to indicate a word (16-bit) or dword (32-bit) size.

11.6.2 Assembler operand types

Every explicit operand to an assembly language instruction is one of:

1. A register, which includes 8-bit, 16-bit, or 32-bit registers, most often a GPR (General Purpose Register), part of a GPR, or a segment register
2. Memory, possibly addressed using none to two registers and/or a displacement, possibly with a segment override. A memory operand generally has the address surrounded by square brackets '[...]'
3. An immediate, possibly byte, word, or dword sized, where a byte may be sign-extended for certain instructions

11.6.3 A quick overview of most 8086 instructions

ALU 2-operand

add, adc, sub, sbb, and, or, xor, cmp, test

ALU 1-operand

neg, not, inc, dec

Multiplication and division

mul, imul, div, idiv

Shifts and rotates

shl, shr, sar, rol, ror, rcl, rcr

Data movement

mov, xchg

Stack

push, pop, pushf, popf

Branch

jmp, jcc, loop, jcxz, call, int, retn, retf, iret

Flags

cld, stc, cmc, cld, std, cli, sti, lahf, sahf

String

lods*, stos*, scas*, cmps*, movs*

Port I/O

in, out

Special: Addresses and segments

lea, les, lds

Special: Prefixes

repe, repne, lock, es, ss, cs, ds

Special: BCD

daa, das, aaa, aas, aam, aad

Special

nop, xlatb, cbw, cwd, hlt

11.6.4 ALU 2-operand instructions

add

Add $op1 + op2$ and store in $op1$

adc

Add $op1 + op2 + CF$ and store in $op1$

sub

Subtract $op1 - op2$ and store in $op1$

sbb

Subtract $op1 - op2 - CF$ and store in $op1$

and

Bitwise AND both ops and store in $op1$

or

Bitwise OR both ops and store in $op1$

xor

Bitwise XOR both ops and store in $op1$

cmp

Calculate $op1 - op2$, affects only flags

test

Calculate bitwise AND, affects only flags

11.6.5 ALU 1-operand instructions

neg

Negate operand in two's complement

not

Bitwise NOT the operand

inc

Increment operand, preserves CF

dec

Decrement operand, preserves CF

11.6.6 Multiplication and division

mul

Multiply to doublewidth result

imul

Multiply signed to doublewidth result

div

Divide doublewidth number

idiv

Divide signed doublewidth number

11.6.7 Shifts and rotates

shl

Shift left

shr

Shift right, fill zeroes

sar

Shift arithmetic right, fill sign bit

rol

Rotate left

ror

Rotate right

rcl

Rotate with Carry left

rcr

Rotate with Carry right

11.6.8 Data movement

mov

Copy data value

xchg

Exchange two data values

11.6.9 Stack

push

Push from operand

pop

Pop into operand

pushf

Push flags

popf

Pop flags

11.6.10 Branch

jmp

Unconditional jump

jcc

Conditional short jump

loop

Decrement counter register and jump if nonzero

jcxz

Jump if counter register is zero

call

Push next instruction's address and jump to subroutine

int

Invoke interrupt handler

retn

Return near

retf

Return far

iret

Interrupt return

11.6.11 Flags

clc

Clear CF (NC)

stc

Set CF (CY)

cmc
Complement CF

cld
Clear DF (UP)

std
Set DF (DN)

cli
Clear IF (DI)

sti
Set IF (EI)

lahf
Load AH from flags

sahf
Store AH to flags

11.6.12 String

lods*
Load from string

stos*
Store to string

scas*
Compare accumulator register to string

cmps*
Compare two strings

movs*
Copy from string to string

11.6.13 Port I/O

in
Input from port

out
Output to port

11.6.14 Special: Addresses and segments

lea

Load Effective Address (calculate offset into register)

les

Load offset and Extra Segment from far pointer in memory

lds

Load offset and Data Segment from far pointer in memory

11.6.15 Special: Prefixes

repe

Repeat string instruction (while ZF set)

repne

Repeat string instruction (while ZF clear)

lock

Lock bus for atomic RMW

es

Override with Extra Segment

ss

Override with Stack Segment

cs

Override with Code Segment

ds

Override with Data Segment

11.6.16 Special: BCD

daa

Decimal Adjust after Addition

das

Decimal Adjust after Subtraction

aaa

ASCII Adjust after Addition

aas

ASCII Adjust after Subtraction

aam

ASCII Adjust after Multiply

aad

ASCII Adjust before Division

11.6.17 Special

nop

Do nothing

xlatb

Load byte from lookup table

cbw

Convert signed byte to word

cwd

Convert signed word to dword

hlt

Halt until an IRQ arrives

Section 12: Variable Reference

12.1 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`

12.2 MMX registers - MMxy

If MMX is available, the debuggee's MMX registers can be accessed as variables. However, as the debugger only supports 32-bit numbers, only half of a 64-bit MMX register can be accessed. The 'x' is a digit from 0 to 7. The 'y', if present, is one of the following letters:

L

Access only low 32 bits

H

Access only high 32 bits

Z

Read low 32 bits, write full 64 bits with zero extension to qword

S

Read low 32 bits, write full 64 bits with sign extension to qword

Letter absent

Same as letter Z

12.3 Options

12.3.1 DCO - Debugger Common Options

DCO1 (alias DCO) to DCO7. Dword. Writable.

12.3.2 DCS - Debugger Common Startup options

DCS1 (alias DCS) to DCS7. Dword. Read-only.

12.3.3 DIF - Debugger Internal Flags

DIF1 (alias DIF) to DIF7. Dword. Read-only.

12.3.4 DAO - Debugger Assembly Options

Dword. Writable.

12.3.5 DAS - Debugger Assembly Startup options

Dword. Read-only.

12.3.6 DPI - Debugger Parent Interrupt 22h

Alias DPRA. Dword. Read-only. Always a 86M segmented pointer. 0 if in TSR mode, or loaded as a device driver, or in bootloaded mode.

12.3.7 DPR - Debugger PRocess

Alias DPSP. Word. Read-only. Always a 86M segment.

12.3.8 DPP - Debugger Parent Process

Alias DPARENT. Word. Read-only. Always a 86M segment. 0 if in TSR mode, or loaded as a device driver, or in bootloaded mode.

12.3.9 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.) This variable does not exist on non-DPMI IDebug builds.

12.3.10 DPSPSEL - Debugger PSP Segment/Selector

The debugger's PSP segment while in 86 Mode, a selector pointing to the same base while in Protected Mode. This variable exists even on non-DPMI builds, where it is always the same as DPSP.

12.4 Default step counts

PPC

Proceed command (section 10.33) default step count

TPC

Trace/Proceed command (section 10.49.1) default step count

TTC

Trace command (section 10.49) 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 10.1. If one of these is set to zero then it is an error to not specify a count explicitly for the corresponding command.

12.5 Default lengths

DEFAULTDLEN

Word. Default length of D/DB/DW/DD commands in bytes (section 10.12). Only used if DEFAULTDLINES is zero. Default is 128.

DEFAULTDLINES

Word. Default length of D/DB/DW/DD commands in lines (section 10.12). Not used if zero. This is a word variable, but setting it to a value higher than 7FFFh is invalid. Default is zero.

DEFAULTULEN

Word. Default length of U command in bytes (section 10.52). Only used if DEFAULTULINES is zero. Default is 32.

DEFAULTULINES

Word. Default length of U command in lines (section 10.52). Not used if zero. This is a word variable, but setting it to a value higher than 7FFFh is invalid. Default is zero.

12.6 Limits

12.6.1 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.

12.6.2 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.

12.6.3 RCLIMIT - RC buffer execution command limit

Doubleword. Default is 4096. If this many commands are executed from the RC buffer, the execution is aborted.

12.6.4 RCCOUNT - RC buffer execution command count

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

12.7 Return Codes

12.7.1 RC - Return Code

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

12.7.2 ERC - Error Return Code

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

12.8 Addresses

12.8.1 A address (AAS:AAO)

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

12.8.2 D address (ADS:ADO)

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

12.8.3 Address behind R disassembly (ABS:ABO)

12.8.4 U address (AUS:AUO)

Default address for the disassembler.

12.8.5 E address (AES:AEO)

Default address for memory entry.

12.8.6 DZ address (AZS:AZO)

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

12.8.7 D\$ address (ACS:ACO)

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

12.8.8 D# address (APS:APO)

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

12.8.9 DW# address (AWS:AWO)

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

12.8.10 DX address (AXO)

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

12.9 I/O configuration

12.9.1 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.

12.9.2 IOC - I/O Columns

Byte. Default 1. Sets the number of columns of the terminal used by BIOS input or DOS input if `getinput` is enabled. 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 (eg by using `INSTALL GETINPUT`). A value between 2 and 39, inclusive, is not recommended.

12.9.3 IOCLINE - I/O Columns for splitting lines in `getinput`

Byte. Default 0. Sets the number of columns of the terminal at which to split overlong lines after the user submits an input line with the `getinput` line editor. If this is zero, splitting overlong lines is disabled and it is assumed that the terminal will split lines as desired. Otherwise, overlong lines (ie those extending past the IOC width) will be split to have no more than the amount of bytes specified by the IOCLINE variable. Makes most sense to set this equal to IOC or DSC, or else to zero. Any nonzero value is allowed.

12.9.4 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.

12.9.5 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.

12.9.6 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.

12.9.7 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.

12.10 I/O reading variables

IOK - Read a keypress

Returns a keypress scancode and/or ASCII text byte. The keypress is read from the current debugger terminal. The terminal waits until a keypress arrives, if none is buffered yet. The keypress is consumed. The low byte is an ASCII codepoint or zero. The high byte is an int 16h scancode or zero.

IOI - Check for input

Returns a nonzero value if input is available to the current debugger terminal. The input is not consumed however. When using DOS, the value 0FFFFh indicates input is available. When using int 16h, a nonzero return will consist of a scancode in the high byte and/or an ASCII codepoint in the low byte. When using serial I/O a nonzero value will be an ASCII codepoint.

12.11 Serial configuration

12.11.1 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.

12.11.2 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. A value between 2 and 39, inclusive, is not recommended.

12.11.3 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.

12.11.4 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>.

12.11.5 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.

12.11.6 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.

12.11.7 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.

12.11.8 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.

12.11.9 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.

12.11.10 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.

12.12 Timer configuration

These variables control some details of the debugger's timers used for waiting in a few places. The affected timers are:

- SLEEP command
- KEEP prompt timeout during serial I/O connection
- Serial output send wait if buffer full

Unaffected timers include:

- DCO3 flag 0400_0000h (delay for a tick before writing breakpoints), this only waits until one tick change is observed
- Interrupt 8 Control pressed check, this does not count tick changes but rather counts interrupt calls

12.12.1 GREPIDLE - getc repeat idle count

Byte. Default 0. If the getc function invokes the idle handling, it will repeatedly call the idle function if this variable holds a nonzero value. The value specifies the amount of repetitions past the first call. This variable is intended for debugging.

12.12.2 SREPIDLE - Sleep repeat idle count

Byte. Default 0. If the SLEEP command invokes the idle handling, it will repeatedly call the idle function if this variable holds a nonzero value. The value specifies the amount of repetitions past the first call. This variable is intended for debugging.

12.12.3 SMAXDELTA - Maximum encountered delta ticks

Word. This value is set anew whenever a wait handler has found a delta ticks larger than the prior value of this variable. After any wait iteration the variable inevitably will be nonzero.

12.12.4 SDELTALIMIT - Delta ticks limit

Word. Default 5. This variable specifies the upper limit of the delta between tick low words accepted as being accurate.

Note that at midnight the tick low word goes from 00AFh or 00B0h to 0000h. The delta appears to be FF51h or FF50h ticks at that point. Therefore the limit should be set small enough that the total wait time is not majorly skewed at midnight.

However, it is possible that the idle handling takes so long that more than one tick has actually gone by until the wait handling is run again. For such system setups it can be desirable to set the limit to more than 1.

A good compromise is to set the limit to between 1 and 6, inclusive. A limit of 6 ticks only skews for 1/3 of a second at midnight.

12.13 **_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.

12.13.1 **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.

12.13.2 **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.

12.13.3 **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 TLA7 counters gives one linear address for getlinear fault injection testing.

12.13.4 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 TSC7 counters gives one counter for getsegmented fault injection testing.

TSA - Test getSegmented Address

Doubleword. Each of the TSA0 to TSA7 counters gives one linear address for getsegmented fault injection testing.

12.14 _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.

12.14.1 MT0 - Mask Test 0

Doubleword. Default 0. Mask AA55_AA55h.

12.14.2 MT1 - Mask Test 1

Doubleword. Default 0011_0022h. Mask 00FF_00FFh.

12.15 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.

12.15.1 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.

12.16 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.

12.17 PSP variables

All of these are read-only. All of them are zero if in bootloaded mode.

12.17.1 PSP - Process Segment Prefix

Word. Always a segment,

12.17.2 PPR - Process PaRent

Alias PARENT. Word. Always a segment.

12.17.3 PPI - Process Parent Interrupt 22h

Alias PRA. Dword. Always a 86 Mode segmented address.

12.17.4 PSPSEL - PSP segment or selector

Alias PSPS. Word. Segment or selector according to mode.

12.18 SR variables - Search Results

12.18.1 SRC - Search Result Count

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

12.18.2 SRS - Search Result Segment

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

12.18.3 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.

12.19 Access variables

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

12.19.1 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.

12.19.2 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.

12.19.3 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.

12.19.4 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.

12.20 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.

12.21 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.

12.22 Rlxy - Real 86 Mode Interrupt vectors

These read-only variables provide access to the 86 Mode interrupt vectors in a more accessible format.

The xx must be a single or two hexadecimal digits, or an index expression in parentheses which evaluates to a number below 256.

The y must be one of the following letters:

- P - Read vector as a 16:16 segmented pointer (dword), ready for use as a pointer-type expression. (Actual use as a pointer-type expression still requires a PTR type keyword.)
- S - Read vector's segment. (Word.)
- O - Read vector's offset. (Word.)
- L - Read vector as a linear address. (3byte.)

12.23 FL.xF - Flag status

These read-only variables support reading the flag status of certain flags.

The x must be one letter to form one of the following names:

CF

Carry Flag

PF

Parity Flag

AF

Auxiliary carry Flag

ZF

Zero Flag

SF

Sign Flag

IF

Interrupt Flag

DF

Direction Flag

OF

Overflow Flag

Note that the FL.xF flags are read-only. To write to a flag with the R command, it is valid to specify just xF as the variable name, but this is not valid within expressions to read the flag status. (It cannot be supported in expressions because some of the flag names are all hexadecimal digits, such as CF.)

12.24 HHRESULT - H command result

Dword. Result of most recent H command. (If H twofold operation is used then the addition result is stored in the variable.)

12.25 DARESET - D.A command result

Word. Result of most recent D.A command. This is set to 0FFFFh by a failed D.A command and to the allocated selector by a successful D.A command. This variable is only present in lDebugX.

12.26 XARESET - XA command result

Word. Result of most recent XA command. This is set to 0FFFFh by a failed XA command and to the allocated handle by a successful XA command. This variable is only present if the debugger was built with the _EMS option or the x.eld has been installed.

12.27 INT8CTRL - Interrupt 8 Control pressed detection time

Word. Number of ticks to wait with Control pressed until breaking into the debugger. This variable is only used if the interrupt 8 handler is installed. If the handler detects that Control is pressed continuously for this length of time while not in the debugger then the handler will break into the debugger. Default is set up for about 5 seconds (5 times 18). Set to 0 to disable Control pressed detection.

12.28 Device mode variables

DEVICEHEADER

Dword. Read-only. Gives segmented 16:16 address of device header installed by the debugger. Zero if not in device mode.

DEVICESIZE

Word. Read-only. Gives amount of paragraphs allocated to device. Zero if not in device mode.

12.29 QQCODE - Q command termination return code

Byte. Default 0. This variable is used to set the return code used by the debugger to terminate itself with interrupt 21h service 4Ch. This only happens if the debugger is in application mode and not in TSR mode, or a device mode debugger has been attached to a process.

12.30 TERMCODE - Debuggee termination return code

Word. Debugger sets this value when it is entered from a debuggee having terminated. The value is what interrupt 21h service 4Dh returned.

12.31 DDTEXTAND - Data dump text AND mask

The variable DDTEXTAND is used as a mask to modify the text code of a data dump before display. It defaults to 0FFh. Setting this to 7Fh will mimic MS-DOS Debug's display of text in its data dump, masking off the high bit. In this case the TOP setting of the D command has no effect.

12.32 AMIS variables

12.32.1 TRYAMISNUM

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.

12.32.2 AMISNUM

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 is considered stale.

12.32.3 TRYDEBUGNUM

The TRYDEBUGNUM variable is a writable byte variable. It defaults to 255. It is similar to TRYAMISNUM, however its content is tried first to find another debugger instance. After that the debugger currently will search starting from number 255 down to 0. Unused if it matches the debugger's own currently installed AMIS handler's multiplex number. This is used by the debugger to find the following services provided by another debugger instance:

- Update IISP Header
- Install DPMI entrypoint hook

- Install fault areas

None of these services are detected and used if DCO3 flag 800_0000h is set. (In the past this flag was wrongly documented as applying to the Update IISP Header service only.)

TRYDEBUGNUM is as well used by Extensions for lDebug to find the following services:

- AMISMSG (one or several listing hints in ELD linker or hint.eld)
- AMISOTH (ldmemoth.eld, instnoth.eld, hintoth.eld)
- AMISCMD (inject.eld)

The DCO3 option flags do not affect the ELD use of AMIS services.

12.32.4 DEBUGFUNC

The DEBUGFUNC variable is a read-only word variable. It defaults to 0. When the Update IISP Header function of the debugger is called and a search for another debugger happens, then this variable receives a 0 if no other debugger is found. It receives a value with the low byte equal to 30h if another debugger was found. (The low byte is set up as the AMIS private function number of the Update IISP Header service.) The high byte is equal to the detected multiplex number then.

12.33 COUNT - List length count

Dword. Any successfully parsed COUNT or S command will write the length of the pattern it counted, given in bytes, to this variable.

12.34 RHCOUNT - Count of RH buffer entries

Word. Reads as the current amount of buffered RH mode entries. May be corrupted if RH mode is not currently enabled.

12.35 ELDAMOUNT - Amount of installed ELDs

Word. Reads as the amount of residently installed Extensions for lDebug. This variable is only supported if the amount.eld has been installed. Due to the ELD architecture, this variable will always read as above-or-equal 2.

12.36 CIP - Current CS's EIP or IP

In Protected Mode in lDebugX, this variable is either a word or dword. Which one it is depends on the D bit of the descriptor corresponding to the current CS selector. In Real or Virtual 86 Mode this variable is always a word. In lDebug without DPMI support this variable is also always a word.

12.37 CSP - Current SS's ESP or SP

In Protected Mode in lDebugX, this variable is either a word or dword. Which one it is depends on the B bit of the descriptor corresponding to the current SS selector. In Real or Virtual 86 Mode this variable is always a word. In lDebug without DPMI support this variable is also always a word.

12.38 Boot loading variables

12.38.1 BOOTUNITFLxx

The boot unit flags are 256 partially writable byte variables. Every flag byte corresponds to an int 13h unit. The following flags are defined:

- 01h Force CHS access, do not detect LBA support with 13.41
- 02h Force LBA access, do not detect LBA support with 13.41
- 04h Force use of BPB's CHS geometry, do not detect with 13.08

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

These flags match the low flags of the IDOS iniload query patch site. All the flags are by default pre-initialised to zero, but this can be overridden in two ways.

The source macro file debug.mac contains the following equates and defines:

```
lufForceCHS:equ 1
lufForceLBA:equ 2
lufForceGeometry:equ 4
luf_mask_writable equ lufForceCHS | lufForceLBA | lufForceGeometry

numdef LUF_DEFAULT_DISKETTE, 0
numdef LUF_DEFAULT_HARDDISK, 0
```

The first define is the default value used for all diskette units, that is unit numbers below 80h. The second define is the default value used for hard disk units, that is unit numbers above-or-equal 80h.

Further, as documented for IDOS boot, the query patch site of a bootable IDebug executable can have its default changed during the build or can be patched later. If the highest bit of the active patch site byte (80h) is set then the flags are used to initialise the boot unit flags of the loaded from unit.

The low two flags exactly match the IDebug boot unit flags. The flag 04h differs a little, to IDOS iniload it means to pass along the geometry either detected by or hardcoded in the prior loader. To IDebug, the flag means to probe the boot sector (presumed to be a diskette or superdiskette with a FAT FS BPB). In both cases, an interrupt 13h service 08h call is avoided.

Section 13: Interrupt Reference

13.1 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 entypoints.

`CDebug` can run in debuggable mode (like `DDebug`) or with debuggable mode disabled. If the `cmd3` loop detects a change in the `DCO6` flag 100h then it will toggle debuggable mode to match the flag. This will involve hooking the mandatory handlers or unhooking them (forcibly).

As a special exception, if the debugger detects that it is running on an HP 95LX, then interrupt 6 is never hooked. This supports the different use of this software interrupt by the software or firmware on this type of device.

13.2 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.

13.3 Interrupt 2Fh - Multiplex (DPMI entryptoint)

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 entryptoint. DebugX attempts to hook this service to return its own entryptoint 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.)

13.4 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.

13.5 Interrupt 2Dh - Alternate Multiplex Interrupt

This interrupt hook is optional. Setting the DCO4 flag 8 or running an `INSTALL AMIS` command instructs the debugger to set up this interrupt hook. Clearing the flag or running `UNINSTALL AMIS` 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 entryptoints of the debugger. Second, lDebug's private AMIS functions 30h, 31h, 33h, 40h, 41h, 42h, and 43h. They are described in the next sections.

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 is considered stale.

Note that the AMIS interface is not AMIS-compliant in a few ways:

- The uninstall function returns 00h (not implemented).
- If IDDebug or ICDebug are in debuggable mode, their mandatory handlers will still be listed in the AMIS function 4 interrupt list despite not being installed. (Their downlink fields will contain FFFFh:FFFFh then.)
- If the build option `_CATCHSYSREQ` is enabled then the SysReq hook will not be listed for AMIS function 5. The interrupt 8 Control pressed hook is also not listed.

13.5.1 AMIS private function 30h - Update IISP Header

This function is provided for use by our programs that use AMIS multiplexers and interrupt handler entryptoints with IISP headers. All TSRs (including RxANSI, IClock, SEEKEXT, KEEPHOOK, FDAPM, FreeDOS SHARE) and SHUFHOOK use this function. (The debugger itself also uses this function, if it is provided by another resident debugger.)

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 supported,
 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, first run
 the command "r dco4 or= 8", or "INSTALL AMIS"
 (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.

13.5.2 AMIS private function 31h - Install DPMI entrypoint hook

This function is for use by lDDebugX (or lCDebugX). It instructs lDebugX (or lCDebugX) to install its DPMI entrypoint hook. It is called by the debuggable debugger right before it tries to install its own hook. Non-zero return values in AL indicate the function is supported. A return value of 0FFh in AL indicates success. Other non-zero return values indicate that no hook occurred. Non-DPMI builds of lDebug return zero. This function should be called only while the InDOS flag is zero.

```
lDebugX - Install DPMI hook
INP:      al = 31h
OUT:      al = FFh if installed
          al = FEh..F0h if not installed but call is supported
          al = 00h if not supported
CHG:      -
STT:      not in DOS
```

13.5.3 AMIS private function 32h - Reserved for lDebugX

```
lDebugX - Reserved
INP:      al = 32h
```

13.5.4 AMIS private function 33h - Install fault areas

This function is by default provided by lDebugX (including the variants lCDebugX and lDDebugX) and is for use by lDDebugX as well as lCDebugX (in debuggable mode). It is called when the debuggable debugger is instructed to INSTALL AREAS.

```
lDebugX - Install fault areas
INP:      al = 33h
          dx:bx -> fault area structure of client
OUT:      al = FFh if installed
          al = FEh..01h if not installed but call is supported
          al = 00h if not supported
CHG:      al, bx, cx, dx, si, di, es, ds
REM:      The area structure is defined in the lDebug sources'
          debug.mac file. The first 32 bytes of the structure
          start with a signature word, which is equal to the
          word value CBF9h (encoding the instruction sequence
          of stc \ retf) if the structure is not currently
          installed into any debugger. The remainder of the
          32 bytes, as well as the details of how the first
          two bytes are used otherwise, are private to the
          debugger that provides this service (the server).
          The area structure may be far-called in 86 Mode. The
          only currently defined function (in al) for this call
          is function 00h, which attempts to uninstall the area
          structure which is being called. It is valid for
          either the server or the client to uninstall an
          area structure if they so wish.
          The fields of the structure behind the first 32 bytes
          point to a number of sub-structures and area function
```

lists and area lists. All of these structures are to be accessed using the same segment as the main area structure. They contain linear start and linear end addresses, which the client sets up before it tries to install the areas. The linear start address is also assumed to point to the segment base address which is used as the reference for the area functions and areas. (They do not have to match the offset part actually used to run the code, but the lists must be based on the linear start address.)

13.5.5 AMIS private function 40h - Display message

This function is provided by an ELD hooking into the debugger's AMIS handler. The ELD is installed by running 'ext amismsg.eld install'. The function provides a way to display a single message, of up to 384 Bytes, to the debugger terminal. (Older revisions only allowed a message size up to 128 Bytes.) The message is displayed by the ELD's inject handler, before the next debugger command is read in the cmd3 command loop.

```
lDebug - AMIS message ELD - Display message to debugger terminal
INP:    al = 40h
        dx:bx -> ASCIZ message, will be truncated if > 384 Bytes
OUT:    al = 00h if not supported
        al = FFh if supported and full message stored
        (older revisions unconditionally returned al = FFh)
        al = FEh if supported and truncated message stored
        (truncation may occur at 383 or 384 Bytes of non-NUL text)
```

Note that the address in dx:bx is a segmented 86 Mode address as the AMIS interface operates in Real/Virtual 86 Mode. However, dx was chosen to pass the segment to simplify calling the interface from Protected Mode. PM code must ensure to fill the register with a segment value, not a Protected Mode selector.

This function is used by the ELD linker or hint.eld to pass along offset hints to TracList, refer to section 7.3.1.

13.5.6 AMIS private function 41h - Query message status

This function is provided by the same ELD as function 40h. It allows to query whether a stored message has been displayed yet.

```
lDebug - AMIS message ELD - Query message status
INP:    al = 41h
OUT:    al = 00h if not supported
        al = 01h if supported and no message is stored
        al = 02h if supported and message is still stored (not yet displayed)
```

13.5.7 AMIS private function 42h - Get other link data

This function is provided by an ELD hooking into the debugger's AMIS handler. The ELD is installed by running 'ext amisoth.eld install'. The function exports the debugger's link data for use by "other link" ELDs running in another debugger instance.

```

INP:    al = 42h
OUT:    al = 00h if not supported
        al = FFh if supported
        bx (86M segment) => link tables
        cx (selector) => link tables
        dx (86M segment) => PSP
        si (selector) => PSP
        di -> link info in link tables section

```

If the debugger is without DPMI support or if not in PM, the selector values may be uninitialised or stale.

13.5.8 AMIS private function 43h - Inject a debugger command

This function is provided by an ELD hooking into the debugger's AMIS handler. The ELD is installed by running 'ext amiscmd.eld install'. The function allows to send commands to be injected into the command loop of the debugger.

Multiple commands can be injected back to back. They will be injected in the order of the calls to this function. However, the buffer is of a fixed size. (Currently, 1024 Bytes.) If the buffer is full, an error will be returned to the caller.

```

INP:    al = 43h
        cx = flags, all reserved for now (must pass as 0)
        dx:bx -> message in byte-counted string,
            1 byte length (<= 254)
            N bytes text, length matching the value of length byte
            1 byte Carriage Return (= 13)
OUT:    al = 00h if not supported
        al = 01h if supported, but buffer is full
        al = 02h if supported, but unknown bit set in cx
        al = FFh if successfully stored in buffer

```

Section 14: Service Reference

These are the services called by the debugger.

14.1 Interrupt 10h

Used for output while InDOS, DCO flag 8 set, bootloaded, when 'INSTALL BIOSOUTPUT' was used (DCO6 flag 200h set), or when DCO6 flag 100_0000h set.

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 06h

Scroll up window (used if CLEAR command done while writing to int 10h)

Function 08h

Get video attribute (only used if highlighting)

Function 09h

Set video attribute (only used if highlighting)

Function 0Eh

Teletype output

Function 0Fh

Get video mode and page

14.2 Interrupt 16h

Used for input while InDOS, DCO flag 8 set, bootloaded, or DCO6 flag 100_0000h is set.

Function 00h

Read keypress (wait until keypress available, consume it)

Function 01h

Read keypress (return if no keypress available, retain it if any)

14.3 Interrupt 2Fh

Function 1261h

PTS-DOS: Get first UMCB

Function 1680h

Idle (Release timeslice to multitasker)

Function 1687h

Get DPMI entryptoint (used and hooked by IDebugX)

Function 4300h, 4310h

XMS detection and get entryptoint

Function 4A06h

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

14.4 Interrupt 12h

Called by booted debugger to determine base memory size.

14.5 Protected Mode Interrupt 31h

Used by IDebugX while in Protected Mode.

Function 0000h

Allocate LDT descriptor

Function 0001h

Free LDT descriptor

Function 0002h

Get selector from segment

Function 0003h

Get next selector increment value

Function 0006h

Get segment base

Function 0007h

Set segment base

Function 0008h

Set segment limit

Function 0009h

Set descriptor access rights

Function 000Ah

Create alias descriptor

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 0301h

Call Real/Virtual 86 Mode far function

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

14.6 Protected Mode Interrupt 2Fh

Function 1680h

Idle (Release timeslice to multitasker)

Function 168Ah

Determine whether DOS extender is available.

14.7 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

14.8 Protected Mode Interrupt 25h

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

14.9 Protected Mode Interrupt 26h

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

14.10 Interrupt E6h

Function bx = 0, ax = -1

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

Function ax = 9

Called early in debugger init when running in dosemu2. This call instructs the VM to start its `-input` injection. It is normally only needed when booting the debugger.

14.11 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.

Function 4DD4h, bx = 0

Detect HP 95LX

14.12 Interrupt 13h

Used by the booted debugger to load scripts, ELDs, 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

14.13 Interrupt 19h

Boot load. Used if booting the debugger fails.

14.14 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.

AMIS private functions 30h, 31h, 33h, 40h, 41h, 42h, 43h of lDebug (refer to section 13.5)

14.15 Interrupt 25h

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

14.16 Interrupt 26h

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

14.17 Interrupt 21h

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

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 1Ah

Set DTA (used by list.eld)

Function 25h

Set interrupt vector

Function 29h

Parse filename

Function 2Fh

Get DTA (used by list.eld and dtadisp.eld)

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 W command

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 4Eh

SFN Find First (used by list.eld)

Function 4Fh

SFN Find Next (used by list.eld)

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 714Eh

LFN Find First (used by list.eld)

Function 714Fh

LFN Find Next (used by list.eld)

Function 71A0h

Get LFN volume information

Function 71A1h

LFN Find Close (used by list.eld)

Function 7305h

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

14.18 Interrupt 67h

EMS services. Used by X commands.

Section 15: Extensions for IDebug reference

Extensions for IDebug (ELDs) can be loaded using the EXT command (section 10.19). Some ELDs operate in a transient way only; some memory is allocated to them when they load and they free this memory again after their run is done. Other ELDs can be installed residently and will reserve some of their memory as used beyond their initial run.

15.1 LDMEM - Dump IDebug memory use.

Displays information on memory use of the debugger. Can be installed residently with INSTALL parameter to provide the LDMEM command, and uninstalled with an LDMEM UNINSTALL command.

The resident LDMEM command as well as the transient ELD provide a number of keywords to control the output:

MEM

Display memory areas of the debugger, with their segments, selectors (in Protected Mode only), and size in Bytes in hexadecimal as well as in KiB.

SEG

Display segments and (in Protected Mode) selectors in a small overview, without size information on the sections.

HISTORY

Display history utilisation, including how many history entries are used, how many history buffer bytes are used and how many there are in total, and the average size per history entry.

STACK

Display stack utilisation, including the total stack size and how many bytes appear to be used (by a heuristic scan).

ELD

Display ELD code instances and data blocks use. Each code instance is displayed either with a code instance name or the indicator 'Free space'. For each code instance, the code start, stop, and size is displayed in hexadecimal, as well as the size in decimal Bytes/KiB. For code instances marked as free, the code instance name is listed behind the size. This may be stale. For code instances which have an ELD data block allocated to them, the data start, stop, and size is displayed on a second line. Finally, if the ELD code buffer or the ELD data buffer have trailing free space, it is displayed as 'Free space' at the end.

ELDVAR

Display ELD variables. The ext variables format, structure size, amount used, and amount total are displayed. For each allocated variable, its flags, maximum array index, name, and entrypoint into the ELD are displayed. The entrypoint display includes the ELD code instance base address and the ELD code instance name.

COMMANDHANDLER

Display ELD command and preprocess handlers. Each entrypoint is displayed with its offset, the offset of its ELD code instance, and the ELD code instance name.

AMISHANDLER

Display ELD AMIS handlers. This is alike the command handlers. (Note that AMIS handlers are always called in Real/Virtual 86 Mode.)

ASMHANDLER

Display ELD assembly handlers. This is alike the command handlers.

PUTSHANDLER

Display ELD puts-related handlers. This is alike the command handlers.

INJECTHANDLER

Display ELD inject handlers. Unlike the command handlers, inject handlers do not point to a chain of handlers. Instead, there is only ever none or one single inject handler installed. That means the display of inject handlers is not very useful for the LDMEM ELD. However, the LDMEMOTH ELD may make this display more useful.

Some keywords are expanded to a list of other keywords:

ALL

All output.

ALLNOSEG

All output, except for the SEG output.

ELDALL

All ELD-related output.

ELDHANDLERS

All ELD handlers output.

The keyword 'HELP' is handled specifically to display an online help page when specified to the transient ELD. It is not valid for the resident ELD's command, however.

If no keywords are specified, the default keyword is assumed as 'ALLNOSEG'.

15.2 HISTORY - Command history utility.

Lists the command history of the debugger, or clears it. Can be installed residently with `INSTALL` parameter to provide the `HISTORY` command, and uninstalled with a `HISTORY UNINSTALL` command.

Provides two subcommands:

`SHOW`

Dump all history entries, in chronological order.

`CLEAR`

Clear the history, deleting all existing entries.

If the transient ELD is run without a keyword, then the online help is displayed.

15.3 DI - Dump Interrupt vectors.

Re-creation of the `DI` and `DIL` commands of the debugger. This allows to use the `DI` commands when the debugger is built with the `_INT=0` build option. Can be installed residently with `INSTALL` parameter to provide the `DI` command, and uninstalled with a `DI UNINSTALL` command.

Refer to section 10.13.

15.4 DM - Dump MCBs.

Re-creation of the `DM` command of the debugger. This allows to use the `DM` command when the debugger is built with the `_MCB=0` build option. Can be installed residently with `INSTALL` parameter to provide the `DM` command, and uninstalled with a `DM UNINSTALL` command.

Refer to section 10.14.

As a special addition, the `DM` command implemented by the ELD additionally allows the following parameters:

`HEADER`

Display a header line first, indicating the meaning of the columns.

`TABLE`

Similar to `HEADER`, but expand the columns to a table format for easier readability.

15.5 RN - Display FPU registers.

Re-creation of the `RN` command of the debugger. This allows to use the `RN` command when the debugger is built with the `_RN=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the `RN` command, and uninstalled with an `RN UNINSTALL` command.

Refer to section 10.40.

15.6 RM - Display MMX registers.

Re-creation of the RM command of the debugger. This allows to use the RM command when the debugger is built with the `_RM=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the RM command, and uninstalled with an `RM UNINSTALL` command.

Refer to section 10.39.

15.7 X - EMS commands.

Re-creation of the X commands of the debugger. This allows to use the X commands when the debugger is built with the `_EMS=0` build option (as is the default now). Can be installed residently with `INSTALL` parameter to provide the X commands, and uninstalled with an `X UNINSTALL` command.

Refer to section 10.57.

15.8 DX - Dump Extended memory.

Re-creation of the DX command of the debugger. This allows to use the DX command when the debugger is built with the `_DX=0` build option (which is the default as of release 9). Can be installed residently with `INSTALL` parameter to provide the DX command, and uninstalled with an `DX UNINSTALL` command. This ELD requires a 386+ machine.

15.9 INSTNOUN - Operate on INSTALL flag nouns.

Displays information on install flags of the debugger. These are the nouns accepted by the `INSTALL` and `UNINSTALL` commands. Can be installed residently with `INSTALL` parameter to provide the `INSTNOUN` command, and uninstalled with an `INSTNOUN UNINSTALL` command.

The operation that can be selected is:

QUIET

Flag for subsequent operation to not display any output.

SET

Enable/install a flag.

CLEAR

Disable/uninstall a flag.

TOGGLE

Toggle a flag.

After the operation, the name of one or more flags must be listed.

If no operation is specified, all flags are listed. The format is as follows for the first name of a flag:

- Flag address in debugger entry section. '0000' if a special flag.
- Flag mask, or the address of the special flag entriypoint. A mask with more than one bit set indicates a reverse mask.
- A single symbol representing the state of the flag: '+' if enabled, '-' if disabled, '?' if a special flag.
- The name of the flag.
- The description of the flag.

For subsequent alias names of a flag, only the name is listed.

15.10 RECLAIM - Reclaim unused ELD memory.

Transient utility to reclaim unused space in the ELD code buffer and the ELD data blocks buffer. This is no longer needed because the debugger now includes the implementation of this tool and automatically reclaims memory before loading an ELD.

This ELD may still be needed in conjunction with ELDCOMP to reset the ELD buffers the way ELDCOMP expects them after a run.

15.11 ELDCOMP - Compare ELDs with differing linker options.

A tool to compare an ELD with its XLD counterpart. XLD is the filename extension typically used to hold a build of an ELD with some linker optimisations. ELDCOMP allows to compare the two, helping to identify and locate relocation errors in the ELD to be tested.

Before running ELDCOMP, you may want to install `nofreeoneshot` and `uninstall paging` to avoid interfering with the operation of ELDCOMP. In addition, it may be useful to run the debugger with the `/X` switch and particularly the `/Y` switch to increase the size of the ELD buffers that may be needed by ELDCOMP.

Up to 8 commands are passed to ELDCOMP, separated by semicolons. The first and fifth command are typically EXT commands. The first one loads an XLD, whereas the fifth loads the corresponding ELD in the same way. The second and sixth command may invoke a residently installed XLD/ELD, and are typically the same. The third and seventh command are used to uninstall a residently installed XLD/ELD, if any, and are typically the same as well. The fourth and eighth command should be `ext reclaim.eld`.

The output of the first pass of ELDCOMP is as follows:

- Output of running the eight commands
- A line with the data length, data hash, code length, code hash of the first run
- A line with the lengths and hashes of the second run
- The line "Data length mismatch" if applicable
- The line "Data hash mismatch" if applicable
- The line "Code length mismatch" if applicable
- The line "Code hash mismatch" if applicable

If any of the mismatch messages were displayed, ELDCOMP will attempt to run its second pass. If the ELD data space doesn't suffice for running the second pass, the message "Not enough ELD data space left!" is displayed. You may want to retry with a debugger running with the /Y switch.

The output of the second pass of ELDCOMP starts out the same as the first pass's. However, the hashes are expected not to equal those of the first pass because the ELD data block lives at another offset. The second pass will additionally display one or both of the following parts:

- "Comparing data", followed by the left, right, and length numbers. The output of a C command follows, with the in-ELD offsets prepended and appended.
- "Comparing code", in the same way.

The offsets displayed at the beginning and end of each line of the C commands are offsets within the ELD data block or the ELD code instance. They can be referenced in the ELD's listing file to find the source of the mismatch.

15.12 AFORMAT - Format assembly output.

Once installed, this ELD hooks into the assembler. After a line is submitted to the assembler, the AFORMAT ELD will dump in hexadecimal the bytes written by the assembler.

15.13 AMISMSG - Display message received on AMIS interface.

This ELD hooks into the debugger's AMIS interface. It provides the AMIS functions 40h and 41h to send messages to the debugger terminal. (Refer to section 13.5.5 and section 13.5.6.) A message may consist of up to 127 bytes of text. The message is displayed either on the next command being read in the cmd3 command input loop using a command injection handler, or when the command 'AMISMSG DISPLAY' is run.

This ELD is intended to be used in the outer debugger to receive the TracList offset hints sent by hint.eld or by the ELD linker of an ELD being loaded in the inner debugger.

15.14 AMOUNT - Provide ELDAMOUNT variable.

This ELD once installed provides the ELDAMOUNT variable. This variable can be read to obtain the amount of installed ELDs.

15.15 BASES - Convert between different numeric bases.

A converter for different numeric bases. Can accept a numeric parameter to be evaluated by the expression evaluator. Using a BASE= specifier or the name of a base (HEXADECIMAL, DECIMAL, BINARY, OCTAL) the input number may be specified as a literal, which accepts unsigned numbers of up to 64 bits. Output is in the four known bases, formatted to resemble the expression evaluator's literal input format. Output can be in one arbitrary base using a trailing OUTPUT keyword, followed by BASE=, GROUP=, and WIDTH= specifiers. The BASES ELD can be installed residently using an INSTALL parameter to provide the resident BASES command.

15.16 CO - Copy debugger terminal output to a file.

Installs the COPYOUTPUT commands using the INSTALL parameter. Once a file is specified

with 'COPYOUTPUT NAME filename' the debugger opens the file to append to it. While not InDOS all output to the debugger terminal is written to the opened file.

Several commands are provided by this ELD:

COPYOUTPUT NAME filename

Set filename and open/create file to append. An existing file is opened for writing, not truncated.

COPYOUTPUT ON

Enable writing to an opened file. (Default.)

COPYOUTPUT OFF

Disable writing to an opened file.

COPYOUTPUT TOGGLE

Toggle the ON / OFF status.

COPYOUTPUT GETINPUTMODE [ALL|FULLONLY|TOGGLE]

Display or change getinput mode of the COPYOUTPUT ELD.

COPYOUTPUT TRUNCATE

Truncate the currently opened file to zero bytes.

COPYOUTPUT CLOSE

Close the currently opened file.

COPYOUTPUT UNINSTALL

Close a file if any is opened, and uninstall the ELD.

The getinput mode determines how getinput output is written to the COPYOUTPUT file:

ALL

All getinput output is written, including Carriage Returns and blanks and printable text that is used to reposition the cursor or redraw the visible text.

FULLONLY (default)

Only the prompt and the final accepted input line are written. Repositioning and redrawing output is filtered out and not written.

15.17 CONFIG - Access debugger config paths.

Allows to show or set the debugger config paths. This ELD operates as a transient ELD only. Run as:

SHOW CONFIG

Show debugger config path.

SET CONFIG path

Set debugger config path.

SHOW SCRIPTS

Show debugger scripts path.

SET SCRIPTS path

Set debugger scripts path.

15.18 DTADISP - Displays the current DOS Disk Transfer Address.

This ELD runs as a transient ELD only. It is only valid to run this ELD when DOS is available.

15.19 IFEXT - Conditionally run a command if an ELD is installed.

Allows to run another command conditionally, using a command of the form `IF [NOT] EXT "extension name" THEN command`. May be used as a transient ELD or installed residently using an `INSTALL` command to provide the `IF EXT` commands.

The extension name to provide must match an ELD code instance name as displayed by a command like `'ext ldmem.eld eld'`. An ELD is considered installed if at least one non-free ELD code instance matches the specified name. The name is matched insensitive to capitalisation. The name must be specified with quote marks if it contains a blank. Names must not be longer than 8 bytes.

For transient use, the `'IF [NOT] EXT'` construct follows after the ELD filename. For instance, `'ext ifext.eld if ext "amiscmd" then r'` will run an R command if the AMIS command ELD is installed.

15.20 KDISPLAY - Displays the current K/N command buffers' content.

This ELD runs as a transient ELD only.

15.21 LIST - List ELD/SLD files, description lines, sizes, help.

Displays the description lines for ELD files or SLD files that are specified with a single pathname pattern. The pattern may contain wildcards in the last component. After the pathname, several keywords may be specified:

- `VERBOSE` to display technical details,
- `HELP` to display the help,
- `SFN` to force use of the DOS SFN find interface instead of trying the DOS LFN find interface,
- `LIB` to recurse into library ELDs displaying the same infos for every embedded ELD.
- `/name` (where the name is a pattern that can contain question marks or a trailing asterisk) to filter library ELDs which match the name pattern. This allows to inspect one or a few of the embedded ELDs without listing all embedded ELDs.

15.22 PRINTF - Print formatted output.

Allows to print formatted output. Can be installed residently or used as a transient tool. The first parameter is a quoted string. Subsequent parameters provide data to format.

Supported escape codes:

- \b Backslash (8)
- \t Tab (9)
- \n Line Feed (10). Usually wanted in a sequence \r\n
- \r Carriage Return (13)
- \s Blank (32)
- \f Form Feed (12)
- \xNN Byte value in one or two hexadecimal digits

Supported format codes:

- %% Literal percent sign
- %X, %x Hexadecimal number (Capitalised, uncapitalised)
- %U Unsigned number
- %D, %I Signed number
- %C Text byte
- %S Text string
- %B Format as Byte/kB/KiB size

Text strings are provided in one of three forms:

- START list-parameter END (refer to section 8.5)
- RANGE range-parameter
- ASCIZ range-parameter

15.23 SET - Access environment variables.

Allows to access the environment block to read or write variables. Can install a resident SET command using an INSTALL parameter, or run transiently using a RUN parameter.

Running the command, the following parameters are accepted:

(None)

Display all variables in order of their occurrence.

variable

Display the content of the given variable, or a message indicating the variable is not set.

variable=

Delete a variable if it exists.

variable=content

Set or modify a variable, if it fits in the environment block.

/E variable=command

Set a variable to the first line of output displayed by the command. The output is not written to the debugger terminal.

/E /xxx variable=command

Set a variable to a line of output displayed by the command. The 'xxx' indicates which line to choose; '/0' chooses the first nonempty line, '/1' the second line, etc.

15.24 USEPARAT - Display disassembly separators.

Installs an output hook into the debugger to display an underscore line after disassembling near or far jumps or near, far, or interrupt return instructions. An equals sign line is displayed after disassembly of a DOS termination call, if it is detected.

Once installed, the command 'USEPARAT WIDTH expression' is provided. It sets the width of the lines to display, between 0 and #80. The default is #39.

15.25 VARIABLE - Expand environment variables.

Installs a command preprocessor hook to expand '%VARIABLE%' specifications in commands. This ELD can be used as a transient ELD or as a resident ELD. Once installed, variable uses in commands are expanded. Note that the command to uninstall the resident variable.eld is 'VARIABLES UNINSTALL', not matching the filename of the ELD.

To use the ELD transiently, a parameter reading either RUN or DISPLAY is to be used. RUN will run the subsequent command with variable uses expanded. DISPLAY will instead display the subsequent text with variable uses expanded.

15.26 WITHHDR - Run commands with temporary DCO flags set.

Installs a prefix command called WITH. This can be used as WITH HEADER or WITH TRAILER to temporarily set DCO flags to enable D command headers or trailers. The ELD can also be used as WITH NODUMP to temporarily set a DCO flag to disable the S command data dump from after S command search results. Command injection is used to reset the flags afterwards.

15.27 AMISCMD - Run commands received on AMIS interface.

This ELD hooks into the debugger's AMIS interface. It provides the AMIS function 43h to inject commands into the debugger. (Refer to section 13.5.8.) This requires this debugger to have run 'install amis'.

In another debugger instance, inject.eld can be used to inject commands into this debugger.

15.28 AMISOTH - Provide other link info on AMIS interface.

This ELD hooks into the debugger's AMIS interface. It provides the AMIS function 42h to export the debugger's link info as an "other link". (Refer to section 13.5.7.) This requires this debugger to have run 'install amis'.

In another debugger instance, "other link" ELDs (ldmemoth.eld and instnoth.eld) can be used to access this debugger's data.

15.29 AMITSRS - List currently installed AMIS multiplexers.

Port of Ralf Brown's AMIS TSR lister. This ELD can be installed residently or used transiently. If the parameter is not equal to INSTALL, then the following parameters are accepted:

VERBOSE

Display all information for each multiplexer.

MPX

Include the line that indicates the multiplex number.

HOTKEYS

KEYS

Include the display of installed hotkeys.

INT

INTLIST

INTERRUPTS

Include the interrupt list display.

VERSION

Include the line that indicates the version number.

ENTRY

Include the line that indicates the private entrypoint.

If no parameters are specified, then only one line is displayed per multiplexer, listing the vendor and product signatures and the description line of each multiplexer.

15.30 BOOTDIR - List directory entries.

List directory entries in bootloaded mode. This ELD runs as a transient ELD only. The first parameter must be a pathname pattern. The last component may include wildcards. The available wildcards are:

Question mark '?'

Matches one byte as a wildcard.

Asterisk ‘*’

Matches any number of bytes as a wildcard, up to the end of the current SFN field.

After the pathname pattern, any number of keywords may follow:

WIDE

Display in wide mode, up to 5 names per line.

SORT

Sort filenames.

DATE

Must be specified along with SORT. Sort by date.

SIZE

Must be specified along with SORT. Sort by size.

REVERSE

Reverse the sort order.

DIRFIRST

Sort directories first before any files.

When sorting, the ELD will detect a buffer for storing the filenames while scanning the directory in the first pass, for sorting the entries. If this buffer overflows, the ELD cannot succeed.

If the date of a directory entry is zero, which would be decoded as "1980-00-00", the date field is blanked out.

15.31 DBITMAP - Dump 8-bit-wide graphics from memory.

This ELD can be installed residently or used transiently.

15.32 DOSCD - Change DOS current directory or drive.

This ELD can only be used transiently. DOS must be available to run this ELD.

If only a drive letter and colon are specified, then the current drive is changed to the indicated drive.

If the first parameter is a keyword ‘BOTH’ then the current drive is changed to the indicated drive and the current directory on that drive is changed to the indicated path. It is an error to specify the keyword ‘BOTH’ if no drive letter is given.

If the first parameter is a keyword ‘IFBOTH’ then if a drive letter is specified, the current drive is changed to the indicated drive. Then the current directory is changed to the indicated path.

The current drive is changed by using interrupt 21h function 0Eh, and checked for success by using interrupt 21h function 19h. The current directory is changed by using interrupt 21h function 3Bh.

15.33 DOSDIR - List directory entries.

List directory entries using DOS. This ELD runs as a transient ELD only. DOS must be available to run this ELD.

The first parameter must be a pathname pattern. The last component may include wildcards. The available wildcards are:

Question mark ‘?’

Matches one byte as a wildcard.

Asterisk ‘*’

Matches any number of bytes as a wildcard, up to the end of the current SFN field. Semantics may differ for LFNs.

After the pathname pattern, any number of keywords may follow:

SFN

Use DOS SFN search, even if LFN search would be available.

WIDE

Display in wide mode, up to 5 names (SFNs) per line.

SORT

Sort filenames.

SORTSFN

Sort filenames, but sort by SFNs rather than LFNs.

SORTLFN

Sort filenames, but sort by LFNs rather than SFNs.

DATE

Must be specified along with SORT. Sort by date.

SIZE

Must be specified along with SORT. Sort by size.

REVERSE

Reverse the sort order.

DIRFIRST

Sort directories first before any files.

When sorting, the ELD will detect a buffer for storing the filenames while scanning the directory in the first pass, for sorting the entries. If this buffer overflows, the ELD cannot succeed.

If the date of a directory entry is zero, which would be decoded as "1980-00-00", the date field is blanked out.

15.34 DOSDRIVE - Get or set a DOS drive.

This ELD runs as a transient ELD only. DOS must be available to run this ELD.

If no parameters are specified, gets the current DOS drive and displays it as the drive letter followed by a colon.

If the parameter 'GET' is specified without a trailing pathname, act as if no parameters are specified.

If the parameter 'GET' is specified with a trailing pathname, parses the pathname and displays its drive letter if any is specified, or else the current drive.

If the parameter 'SET' is specified with a trailing pathname, then the pathname must contain a drive letter. The current drive is changed to the specified drive.

If the parameters 'SET QUIET' are specified, then the trailing pathname and its drive letter specification are optional. If no such drive specification is included, the command silently does nothing.

15.35 DOSPWD - Display DOS current directory.

This ELD runs as a transient ELD only. DOS must be available to run this ELD.

If no parameter is specified, displays the current directory on the current drive.

If a parameter 'NUMBER' is specified, then a trailing plain number parameter is parsed from an expression. This number must be below 32. It specifies the drive of which to get the current directory.

Otherwise the parameter must be a drive letter followed by a colon. It specifies the drive of which to get the current directory.

15.36 EXTNAME - Guess EXT and Y command filename extensions.

Installs residently to guess EXT and Y command filename extensions. Install using the INSTALL keyword.

Once installed, running an EXT or Y command without a filename extension will guess the filename extension. The EXT command will guess the extension as '.ELD' first and '.XLD' second. The Y command will guess the extension as '.SLD'.

The following commands are additionally provided:

EXTNAME WARNEXT

Display status of warning on unknown filename extension.

EXTNAME WARNEXT ON

Enable warning on unknown filename extension.

EXTNAME WARNEXT OFF

Disable warning on unknown filename extension.

EXTNAME WARNEXT TOGGLE

Toggle warning on unknown filename extension.

EXTNAME GUESSEXT

Display status of guessing filename extension.

EXTNAME GUESSEXT ON

Enable guessing filename extension.

EXTNAME GUESSEXT OFF

Disable guessing filename extension.

EXTNAME GUESSEXT TOGGLE

Toggle guessing filename extension.

15.37 INJECT - Inject commands into other debugger instance.

Injects commands into other debugger instance using the other's AMIS function 43h (refer to section 13.5.8). (This function is provided by the AMISCMD ELD, refer to section 15.27.)

15.38 INSTNOTH - INSTNOUN which operates on other link debugger.

INSTNOUN variant (refer to section 15.9) that operates on another debugger instance. This requires the other instance to have installed the AMISOTH ELD (refer to section 15.28) and its AMIS handler to provide the AMIS function 42h (refer to section 13.5.7).

15.39 LDMEMOTH - LDMEM which operates on other link debugger.

LDMEM variant (refer to section 15.1) that operates on another debugger instance. This requires the other instance to have installed the AMISOTH ELD (refer to section 15.28) and its AMIS handler to provide the AMIS function 42h (refer to section 13.5.7).

15.40 LINFO - Display status of L command.

Installs residently to display status of program-loading L command. Running a program-loading L command will then display some information on the file being loaded and the process that was created.

Additionally the following commands are provided:

LINFO FILE

Display the program load filename. This is what an L command will show before the command is processed.

LINFO FILE QUIET

As LINFO FILE, but display nothing if no program load filename specified.

LINFO PROBE

Probe the program load file and current PSP. This is most of what a successful L command will show after the command is processed.

LINFO PROBE QUIET

As LINFO PROBE, but display nothing if no program load filename specified.

15.41 PATH - Path search for K/N commands.

Provides path search and filename extension guessing for the K and N commands. This ELD can be installed residently or used transiently.

If used transiently, the first parameter must be the command letter 'K' or 'N'. The path search is done on the program load filename specified to the given command, and then the command is run with the resulting pathname. A warning is displayed if the WHILE buffer or DOS are not available, as path search will not happen in this case. An error is displayed if the expanded pathname doesn't fit into the `line_in` buffer.

If used residently, the K or N commands are intercepted in the command handler. The path search is done on the program load filename, and if found the command is modified to receive the resulting pathname. *No warning* is displayed if the WHILE buffer or DOS are not available, but path search will not happen in this case. An error is displayed if the expanded pathname doesn't fit into the `line_in` buffer.

The following commands are additionally provided when installed residently:

PATH WARNEXT

Display status of warning on unknown filename extension.

PATH WARNEXT ON

Enable warning on unknown filename extension.

PATH WARNEXT OFF

Disable warning on unknown filename extension.

PATH WARNEXT TOGGLE

Toggle warning on unknown filename extension.

PATH GUESSEXT

Display status of guessing filename extension.

PATH GUESSEXT ON

Enable guessing filename extension.

PATH GUESSEXT OFF

Disable guessing filename extension.

PATH GUESSEXT TOGGLE

Toggle guessing filename extension.

PATH PATHSEARCH

Display status of searching path.

PATH PATHSEARCH ON

Enable searching path.

PATH PATHSEARCH OFF

Disable searching path.

PATH PATHSEARCH TOGGLE

Toggle searching path.

15.42 EXTLIB - Library of ELDs.

Library of ELDs to be used instead of single files. This library contains most other ELDs. To load an embedded ELD, specify its filename or ELD code instance name as a parameter to the EXTLIB ELD. The name may be specified with an ‘.ELD’ or ‘.XLD’ filename extension, which is ignored. All trailing parameters after the name are passed to the embedded ELD.

If no ELD name or the name ‘HELP’ is specified, the EXTLIB ELD's help is displayed instead. If after the ‘HELP’ keyword one more keyword follows, the short or long list of embedded ELDs is displayed along with the EXTLIB help:

WIDE

Display ELD filenames in a wide format, up to 8 ELDs per line.

DESCRIBE

Display ELD filenames along with the corresponding ELD description lines, one ELD per line.

15.43 EXTPAK - Compressed library of ELDs.

This is used exactly the same as extlib.eld (section 15.42) except that the embedded ELDs are compressed using heatshrink compression. This means that operations like the HELP DESCRIBE list or scanning for an ELD code instance name may take longer than with the uncompressed extlib.eld.

15.44 QUIT - Quit the machine.

Quits the machine. Can be installed residently or used transiently. When run with a parameter ‘RUN’ then this ELD will attempt to quit (shut down) the currently running machine or VM. When ‘install quickrun’ has been used, running this ELD without a parameter will also attempt to quit the machine.

15.45 DOSSEEK - Get or set the DOS 32-bit seek of a process handle.

This ELD only operates transiently, and requires DOS to be available.

The first parameter is either GET or SET. The second parameter is an expression that evaluates to a process handle number. This number must refer to a process handle that is opened in the current debuggee process.

For getting the seek, after the process handle there may follow the VAR index specification. If it is given, then the specified V variable (section 12.16) is set to the current 32-bit seek. Further, a QUIET keyword may follow to suppress the output of the command. If not both a VAR and a QUIET keyword are specified, then the output will be like the following example:

```
-ext dosseek.eld get 5
Process handle 5 current seek is 0000_1400 #5120 5.0 KiB
```

For setting the seek, after the process handle another keyword must follow:

ABS

Set absolute seek.

CUR

Set signed offset from current seek.

EOF

Set signed offset from EOF.

After the keyword, an expression follows which gives the 32-bit offset to seek to.

15.46 ALIAS - Define aliases.

Define simple aliases that are replaced in a command preprocess handler. This ELD must be installed residently.

To add an alias, run as 'ALIAS ADD aliasname expansion'.

To delete an alias, run as 'ALIAS DEL aliasname'.

To list aliases, run as 'ALIAS LIST' or 'ALIAS LIST aliasname'.

15.47 DPB - Display a DOS drive's DPB.

Display a DOS drive's DPB (MS-DOS v4 to v6 layout, optionally with FreeDOS, MS-DOS v7.10, or EDR-DOS FAT32 extensions). Can be installed residently with INSTALL parameter to provide the DPB command, and uninstalled with a DPB UNINSTALL command.

An optional parameter specifies the drive to access. It defaults to zero. A drive letter may be specified with a trailing colon. Otherwise, a number is parsed. Zero means to access the DOS current drive. The number one corresponds to drive A:, two to drive B:, and so on. The DPB is accessed using interrupt 21h function 32h, which requires DOS to be available. Enhanced DR-

DOS may return DPBs for FAT32 drives on this function whereas FreeDOS and MS-DOS 7.10 may refuse to do so.

Instead of the drive parameter, the keyword ADDRESS may be specified. It must be followed by an address parameter. This address specifies where to read the DPB from. This address uses a selector in Protected Mode, except when the parameter is prepended by a dollar sign.

Instead of the drive parameter, the keyword LIST may be specified. After this keyword, either the keyword SKIP or the keyword DRIVE has to follow.

After the keywords LIST SKIP a number is parsed. The list of DPBs is detected from the DOS List of Lists and a number of DPBs are skipped in the chain until reaching the target. The number specified after the keyword indicates how many DPBs to skip.

After the keywords LIST DRIVE a drive is parsed. This may be either a number or a drive letter. The list of DPBs is detected from the DOS List of Lists and the DPBs are scanned for one matching the specified drive in its first field.

Between the LIST keyword and the next keyword, an address for the DOS List of Lists may be inserted. This starts with an AT keyword. Then an address parameter is parsed. This address parameter is always a 86 Mode segmented address, even in Protected Mode. The address parameter's segment defaults to the DOS data segment as obtained from reading the 86 Mode interrupt 31h vector's segment. The address specifies where to find the DOS List of Lists.

Before the LIST or ADDRESS keywords or the drive specifier, the keyword EXTENDED= may be specified to display an EDPB. It must be followed by one of the known DOS version keywords. These are: FREEDOS, MSDOS, or EDRDOS. The DOS keyword indicates what format of EDPB to display.

Before the EXTENDED= keyword, the keyword ONLY may be specified. If specified, only the EDPB extended fields are displayed, skipping the base DPB.

As an example, a drive with a typical 1440 KiB diskette file system will display as follows:

```
-ext dpb.eld A:
DOS function 32h call returned al=00h.
DOS function returned 86 Mode address=00D9:1E40
Drive                00: 00  0
DeviceUnit           01: 00  0
BytesPerSector        02: 0200  512
HighestSectorInCluster 04: 00  0
SectorsPerClusterShift 05: 00  0
ReservedSectors       06: 0001  1
AmountFATs            08: 02  2
AmountRootEntries     09: 00E0  224
DataStart             0B: 0021  33
MaximumCluster        0D: 0B20  2848
SectorsPerFAT         0F: 0009  9
RootStart            11: 0013  19
DeviceHeader          13: 0070_0610  7341584
MediaID              17: F0  240
Accessed             18: 00  0
Next                 19: 00D9_1E7D  14229117
```

```
FreeClusterNext          1D: 0000  0
AmountFreeClusters       1F: FFFF  65535
-
```

Note that at least the FreeDOS kernel always resets the `FreeClusterNext` and `AmountFreeClusters` fields whenever the DPB is obtained using interrupt 21h function 32h.

Invalid drives (including file system redirector drives) will result in a nonzero return in AL:

```
-ext dpb.eld z:
DOS function 32h call returned al=FFh.
-
```

15.48 RDumpldx - Dump text bytes pointed to by DS:SI and ES:DI in R register dump.

Once installed, this ELD hooks into the debugger's R command register dump. If the dump is in 16-bit mode and not configured for 40-column display, and IDebugX is not in Protected Mode, this ELD will take effect. It will display a trailer to the first line of the register dump, listing what text bytes are found at the addresses indicated by DS:SI and ES:DI.

Example:

```
AX=5A20 BX=0000 CX=051C DX=0000 SP=FFFE BP=0000 SI=0082 DI=0000  SI='s' D
DS=5A21 ES=5A21 SS=5A21 CS=5A21 IP=04D0 NV UP EI PL ZR NA PE NC
5A21:04D0 AC                      lodsb
```

15.49 RDumpStr - Dump text pointed to by DS:DX in R register dump.

Once installed, this ELD hooks into the debugger's R command register dump. If the dump is in 16-bit mode and not configured for 40-column display, and IDebugX is not in Protected Mode, this ELD will take effect. It will display a trailer to the second line of the register dump, listing what text is found starting at the address indicated by DS:DX.

Example:

```
AX=0A07 BX=FD2B CX=0000 DX=02BB SP=FFFC BP=0000 SI=0087 DI=0000
DS=5A21 ES=5A21 SS=5A21 CS=5A21 IP=03FF NV UP EI PL NZ NA PO NC  "Other ke
5A21:03FF 80FFFD                      cmp      bh, FD
```

15.50 CHECKSUM - Calculate checksum over a memory range.

This ELD can be used as a transient command or installed residently. Install with an `INSTALL` keyword. The `CHECKSUM` command accepts a range parameter, with a default length of 512 Bytes. After the range an optional initial sum can be specified. This sum defaults to 1.

The checksum is an unsigned 16-bit number. It is calculated according to the following pseudocode:

```
initial sum = 1
for every byte in range:
```



```

sum = sum * 31
sum += byte value
sum = sum & 0FFFFh

```

15.51 HINT - Display TracList hints to outer debugger

This ELD can be used as a transient command or installed residently. Install with an `INSTALL` keyword. When run, this command displays TracList listing offset hints for all currently installed ELDs of the debugger running this ELD. The hints are written to the terminal of another IDebug instance, assumed to be the outer debugger. This utilises that outer debugger's `AMISMSG` service, so the `amismsg.eld` and `AMIS` interface must be installed in that debugger.

An optional keyword `SKIPSELF` may be specified. If it is given, then the `hint.eld` will not write a hint corresponding to its own ELD code instance. This is useful for running with `eldcomp.eld`.

15.52 HINTOTH - Display TracList hints of the other link debugger

This ELD can be used as a transient command or installed residently. Install with an `INSTALL` keyword. When run, this command accesses the other link debugger's installed ELDs. This utilises the other link debugger's `AMISOTH` service, so the `amisoth.eld` and `AMIS` interface must be installed in that debugger. The `HINTOTH` command will display TracList listing offset hints for all currently installed ELDs of the detected other link debugger.

15.53 CHSTOOL - Work with int 13h partitions and geometry.

Utilities to work with int 13h unit partitions and geometry. Can be used as a transient command or installed residently.

15.54 S - Search command with additional support for WILD and CAPS keywords

This ELD provides the same basic command as the `S` command (section 10.47). However, it allows some extensions as well. The `S` ELD can be used transiently or installed residently. To invoke a resident `S` ELD the command letter `S` must be recognised as a string, that is the search range's first letter or digit must not immediately follow the '`S`' letter.

The basic command format is:

```
search          S range [REVERSE] [SILENT number] [RANGE [CAPS] range|lis
```

The keywords `REVERSE`, `SILENT`, and `RANGE` are all described in section 10.47. The list parameter is based on the debugger's generic list parameter type, refer to section 8.5. However, it adds three new keywords:

15.54.1 WILD - Search wildcard

This keyword when encountered in the search pattern list inserts one wildcard item of the current element size. As the default item size is byte, a `WILD` keyword defaults to inserting a byte-sized wildcard. After for example `AS WORDS`, a `WILD` keyword instead will insert a word-sized (2-byte) wildcard.

Using wildcards may degrade search performance.

15.54.2 CAPS - Search with capitalisation folding

This keyword enables the search to treat every subsequent byte of the search pattern as requiring capitalisation be folded, so that small letters and big letters (in the ASCII alphabet) will match one another.

The CAPS keyword may appear in a search pattern list, or after the RANGE keyword. In the latter case capitalisation is folded for the entire search pattern range.

Using the CAPS keyword may degrade search performance.

15.54.3 UNCAPS - Reset search to do no capitalisation folding

This keyword is only allowed in a search pattern list. It resets the status set up by the CAPS keyword, so that byte values are matched exactly again.

15.54.4 S ELD internals

The wildcard and caps search modes are implemented using a tag buffer. Every tag consists of two bits, one of which indicates that the corresponding byte is a wildcard and the other one for capitalisation folding.

There are three families of internal functions to handle search that may involve nonzero tags:

- Repeated scan byte for a given value
- Scan a single byte for a given value
- Compare trailing string after an initial byte match was found

15.54.4.1 S ELD - Byte scan functions

The first function is used to scan for the pattern's first byte in almost the entire search space. The second function is used to scan the very last possible match of the first byte. This distinction is needed because of how the debugger handles search space length of up to 64 KiB with a 16-bit counter.

Both of these functions must work either with Direction Flag cleared (UP) or set (DN). Additionally, there is a branch to an a32 variant of each of these functions to allow searching in 32-bit segments.

The repeated scan byte functions are implemented by calling the single byte scan functions in an appropriate loop.

If the tag of the search pattern's first byte is zero, simple functions that only consist of the a32 dispatching and a single string operation are used. This is a speed optimisation.

15.54.4.2 S ELD - Trailing string comparison function

The third function is used after a candidate match is identified from the first byte scan. If any tags are nonzero in the search pattern, then complicated functions are used. These will loop like the single string operation `'repe cmpsb'` would, but internally they use the single byte scan functions to handle tags appropriately. The tags are iterated through alongside the pattern values.

This function always works with the Direction Flag cleared (UP) although it would work

otherwise as well. As for the other two functions there is a32 dispatching for searching in 32-bit segments.

If all tags are zero, the trailing string comparison function is handled by simple functions consisting of a32 dispatching and a single string operation. If the RANGE CAPS keywords are used together, medium complex functions are used which always pass an immediate tag value to the single byte scan function. Otherwise the full handling of all possible tag combinations is done, and the loop keeps track of both the current offset, the remaining length, and the current bit position in the tag buffer.

Section 16: Extension for IDebug format

16.1 ELD executable format

An ELD file may have an "ELD trailer header". If present, this structure must be located at the very end of the file. It has the following structure:

```
        struct ELD_TRAILER_HEADER
eldthSignature:      resb 8   ; "ELD1TAIL"
eldthHeaderOffset:   resd 1   ; negative displacement from eldthSignature
                        ; add this dword to the seek offset
eldthReserved:       resw 1   ; reserved, must be initialised to zero
eldthChecksum:       resw 1   ; sum of all words in trailer header = 0
        endstruct
        endarea ELD_TRAILER_HEADER, 1
```

If the trailer header is present, then the seek offset of eldthSignature plus the value in dword [eldthHeaderOffset] gives the seek offset of the ELD header. The ELD header must start before the trailer header, that is the computed ELD header offset must be below the offset of the ELD trailer header. If the ELD trailer header is not present then the ELD header must be located at seek offset 0, at the very beginning of the file.

(The eldapend.c tool allows to append an ELD to an existing file along with the appropriate ELD trailer header.)

The ELD header structure has the following structure:

```
        ; ELD executable header
        struct ELD_HEADER
eldhSignature:      resb 4   ; "ELD1"
                        resb 3   ; reserved
                        resb 1   ; 26 (Ctrl-Z)
eldhCodeOffset:     resd 1   ; position displacement from eldSignature
eldhCodeImageLength: resw 1   ; amount bytes
eldhCodeAllocLength: resw 1   ; amount bytes, added to prior
eldhDataOffset:     resd 1
eldhDataImageLength: resw 1   ; (zero allowed)
eldhDataAllocLength: resw 1   ; (zero allowed)
eldhCodeEntrypoint:  resw 1
eldhReserved:       resb 6   ; reserved
        endstruct
        endarea ELD_HEADER, 1
```

The first four bytes and the 8th byte are checked to match the known format by the debugger. Extensions should go into the last 6 bytes which should be zero-initialised for now. That is,

unless a new ELD format is chosen, which is indicated by changing the signature in the first four bytes. The header is exactly 32 Bytes long.

The header must be found either using the ELD trailer header or it must be found at the very beginning of the ELD file. The offsets given in the fields `eldhCodeOffset` and `eldhDataOffset` are to be interpreted as being displacements that are added to the base equal to the position of the start of the ELD header structure. (This is the position of the `eldhSignature` field.)

The length fields are typically paragraph-aligned but this is not a requirement. However, the resulting allocation of code and data space in memory is always rounded up to a paragraph boundary. The image and alloc length fields for either the code space or the data space must not overflow 16 bits when added. (Code and data together may exceed 64 KiB however.) Keep in mind that ELD code space is typically 8 or 16 KiB (up to 65_520 Bytes maximum) and ELD data space is by default 16 KiB.

Allocation beyond the images is not generally initialised by the debugger. If the ELD wishes to initialise it, this needs to be included in the ELD code.

The entrypoint field is an offset within the code section. The exact IP value is derived from this by adding the ELD's instance base offset to the entrypoint field value. The CS value points to the ELD code segment or a code selector referencing the same segment. The entrypoint is entered with the following function protocol:

```
    ; INP:  es:dx -> loaded initial ELD image
    ;       ELD instance structure filled
    ;       es => ELD code area
    ;       ds:di -> link info
    ;       ss:bx -> loaded initial data
    ;       ss:si -> command line tail
    ;       cs:ip -> entrypoint
    ;       ss:sp -> far return address for current mode
    ; STT:  UP, EI
```

All other registers (cx, ax, bp, fs, gs, high words) are don't cares.

It is expected that the entrypoint will run a linker, which uses the link info pointed to by DS:DI to link the ELD with the exposed interfaces of the debugger.

16.2 ELD instance format

Each ELD instance is stored in the ELD code space with the following format:

```
    ; ELD instance header
    struct ELD_INSTANCE
eldiStartCode:      resw 1  ; -> this structure itself (para aligned)
eldiEndCode:        resw 1  ; -> behind memory used by instance (para aligned)
eldiStartData:      resw 1  ; -> data in data entry section (para aligned)
eldiEndData:        resw 1  ; -> behind data (para aligned), or 0
eldiIdentifier:     resb 8  ; blank-filled text
eldiFlags:          resw 1  ; flags
```

```

eldiReserved:                resb 14 ; reserved
    endstruc
    ; flags for eldiFlags:
eldifResident:                equ 1

```

The first field references its own address. The second field indicates where the next ELD instance lives, unless it matches the value of word `[extseg_used]`. The second field must always be at least the value of the first field plus 32 (the size of the instance header). The third and fourth field specify where the ELD data block associated with this ELD instance lives, unless both fields are zero indicating no ELD data block. All of these first four fields are initialised by the loader (usually the debugger EXT command handler) upon initialising an ELD. However, the ELD is free to modify them within the given constraints. (Note that the ELD buffer fields described in section 16.6.1 must be modified to match the ELD instance fields, depending on how they are changed.)

The fifth field contains an 8 byte all-printable-ASCII text name that identifies the ELD. It should be initialised within the ELD executable. When shorter than 8 bytes, it should be padded with blanks (value 32).

The flags field indicates whether the ELD instance is resident. If it is resident, then other commands (including the EXT command and ELD space reclaim) will not re-use the code or data areas referenced by this instance. If it is not resident, then any re-entry into the debugger's cmd3 command loop should be assumed to overwrite this instance's memory. Note that any DOS I/O and any debugger code that may branch to the debugger error handlers may become a point at which the debugger returns to the command loop.

The remaining fields are reserved and should be initialised to all zeroes.

16.3 ELD link info format

The ELD link info table looks like the following structure:

```

    ; ELD link info header
    struc ELD_LINKINFO
eldlSignature:                resw 1 ; 0E1D1h
eldlReserved:                resw 2
eldlUseLinkHash:              resw 1
eldlDataAmount:               resw 1
eldlDataPrefixes:             resw 1
eldlDataEntries:              resw 1
eldlDataAddresses:            resw 1
eldlCodeAmount:               resw 1
eldlCodePrefixes:             resw 1
eldlCodeEntries:              resw 1
eldlCodeAddresses:            resw 1
    endstruc

```

The `eldlUseLinkHash` field should be 1 to indicate hashes are used, and 0 to indicate the prefix text is used instead. Other values are not valid to current ELDs.

It is expected that a linker embedded into an ELD will use this table to link to the exposed

debugger interfaces. The linker may also be used to resolve internal references of the ELD to its own code section and data block, both of which are loaded to dynamically chosen offsets.

The amount fields indicate how many different links of each type there are. All three arrays of the same type have as many entries as the amount field of that type indicates.

The prefixes table contains either a 16-bit hash value per link name, or the first two text bytes of each link name. The hash value is calculated using the symbolic branch text hash function, which is implemented as follows:

```
hash = 1
for each text value byte:
    hash = (hash * 31 + value byte) & 0FFFFh
```

The entries table contains a word per link that points to a byte-counted message string in the same segment as the link info table. This message starts with a byte giving the remaining length of the message. If hashes are used, the message is the entire link name. Otherwise, the message is the link name remaining after the first two bytes. (Link names shorter than two bytes are not allowed. Link names should be at most 64 bytes long.)

The address table for the data links contains one word per link. This address is equal to the data link value, typically but not always an offset in the debugger stack segment.

The address table for the code links contains two words per link. The first word is equal to the code link offset value. This is the address of the code in its respective code section. The second word indicates which section the link points to. It must be below 3, and indicates the section as follows:

0

1DEBUG_CODE

1

1DEBUG_CODE2 (only used if _DUALCODE build option enabled)

2

1DEBUG_ENTRY (not used yet)

These values must be used with the contents of the `linkcall_table` to determine the offsets within the default ELD (`LINKCALL`) which must be called in order to transfer control to the respective section. The reference to the link call table must be found by linking part of the linker itself first, using the data link to this table.

16.4 ELD link call table

The link call table contains mappings from section values in the code links to offsets within the ELD code segment. These offsets are entries into the builtin default ELD, called `LINKCALL`. This ELD is installed by the debugger's init if possible. It currently occupies a fixed size of 128 Bytes, including its own ELD instance header stored at offset 0 in the ELD code segment. Thus if no regular ELDs are currently installed residently then a regular ELD will always be loaded to

offset 0080h in the ELD code segment. (Thus a `--local-offset 80h` switch to TracList is useful.)

The link call table has the following format:

word .init

Specifies table is initialised and its format. Must be 1.

word Amount

Specifies how many section records follow. Typically 2 or 3.

dword per Section Record

Each record has the following format:

word Section Index

Match for a section value from a code link.

word Call Offset

Offset to call in ELD code segment.

An ELD link call is constructed as follows:

- Opcode byte for call rel16: 0E8h.
- Displacement word in rel16 format pointing to appropriate call offset.
- 2 bytes of code to handle return path, typically a jump short (0EBh).
- Offset word of code to call in target section.

The ELD linker is responsible for filling in this structure for every link call that is to be used. The displacement is calculated from the call offset field of the link call table entry matching the section value from the code link. It must be calculated by subtracting the offset behind the near call rel16 instruction. The offset word to call is copied from the code link directly.

16.5 ELD linker internals

The ELD linker is composed of three files:

elddata.mac

ELD link data and internal relocations macros

eldcall.mac

ELD link call macros

eldlink.asm

ELD linker code and dump of relocation tables

16.5.1 ELD data macros

The ELD data macros contain the following mmacros for users:

`internalcoderelocation`

Can be used in DATA section or CODE section. Section type must match 'DATA' to indicate a data section. Links a 16-bit word to the place the ELD code section is loaded. Parameter specifies offset from current assembly address, defaulting to -2 to place the relocation at the end of the prior instruction. The relocation must be filled with an address that uses the code section vstart, typically a label in the code section.

`internaldatarelocation`

Can be used in DATA section or CODE section. Section type must match 'DATA' to indicate a data section. Links a 16-bit word to the place the ELD data section is loaded. Parameter specifies offset from current assembly address, defaulting to -2 to place the relocation at the end of the prior instruction. The relocation must be filled with an address that uses the data section vstart, typically a label in the data section.

`linkdatarelocation`

Can be used in DATA section or CODE section. Section type must match 'DATA' to indicate a data section. Links a 16-bit word to a data link of the debugger. First parameter is data link name. Second parameter specifies offset from current assembly address, defaulting to -2 to place the relocation at the end of the prior instruction. Third parameter is the keyword 'required' (default) or 'optional'. The relocation must be filled with an address based on the label `relocateddata`. The value of this label is subtracted during linking, and the value of the data link is added. If the data link is optional, a missing link will have the linker zero the relocation field.

16.5.2 ELD code macros

The ELD code macros contain the following mmacros for users:

`houdini`

Emits a conditional int3 breakpoint. Can be disabled at build time with `_HOUDINI=0` define. Can be disabled at run time by clearing DCO7 flag 100h or disabling debuggable mode.

`extcall`

Sets up a 7-byte ELD extension call into the debugger. This should only be used in the CODE section, that is a section with type not equal to 'DATA'. An appropriate reference to a code link is added to the link tables. There is a second parameter allowed, defaulting to 'required'. The other options are 'optional' and 'PM required'. When the code link for an optional extcall is not found, the near call instruction is overwritten with three NOP instructions.

`extcallcall`

Sets up a 3-byte call to an 8-byte extension call site, which will be composed of a 7-byte ELD call and a `retn` instruction. This should only be used in the CODE section, that

is a section with type not equal to 'DATA'. Multiple `extcallcall` uses to the same target will use the same extension call site. This is the same code size for two calls, and saves code size for three or more calls. However, it deoptimises the call stack because an additional indirection is added. (The extension call site spends another word on the near return address that points back to the `extcallcall` user.) This macro cannot be used after `eldcall_dump_callcall` is used.

`eldcall_dump_callcall`

Must be called like '`eldcall_dump_callcall ELDCALL_CALLCALL_LIST`'. Dumps the extension call sites for all prior uses of the `extcallcall` mmacro. Also will disable further use of the `extcallcall` mmacro. This should be used in the CODE section, before the label to calculate the resident or installed size. (Resident size is the ELD code size after the linker is discarded. Installed size is the ELD code size that remains if the ELD is installed residently.)

16.5.3 ELD linker sources

The ELD linker does not define any mmacros. The assembly source file generally should be included at the very end of the CODE section, and must be placed after all uses of the ELD data and code macros. The only lines after the include directive should be an alignment directive and the equate for the code section size.

The linker uses two labels: 'linker' is the code entrypoint into the linker, which should be called like this:

```
; INP:  es:dx -> loaded initial ELD image
;       ELD instance structure filled
;       es => ELD code area
;       ds:di -> link info
;       ss:bx -> loaded initial data
;       ss:si -> command line tail
;       cs:ip -> entrypoint
;       ss:sp -> far return address for current mode
; STT:  UP, EI
```

This is typically entered into the ELD executable header's field named `eldhCodeEntrypoint`, using a directive like '`dw linker - code`'.

The second label used by the linker is 'start', which the linker will branch to once it has successfully finished linking. This is called with the following protocol:

```
; INP:  es => ELD code segment (writable)
;       ds:si -> command line tail
;       ds:bx -> ELD data block
;       ss:sp -> far return address for current mode
; STT:  ds = ss, UP, EI
```

The far return address passed to these two entrypoints allows to return to the debugger without the need for a code link to the `cmd3` command loop. It expects a result code in `ax`, which it will display as an error code in case it is above-or-equal `0FF00h`. The linker returns with code `0FFFFh` if a required link is missing. The reclaim ELD returns with code `0FFFDh` on errors.

16.5.4 ELD two-pass linker

The linker will use two passes by default. The first pass links the linker itself. The second pass links the ELD application. If the first pass succeeded, the second pass may utilise the debugger output interfaces to display diagnostic messages (warnings or errors). The second pass will thus emit error messages indicating exactly which links are missing, if any.

16.6 ELD interfaces

16.6.1 ELD code and data buffers

The ELD code section is referenced with the following variables:

`extseg`

ELD code section's segment

`extcssel`

In Protected Mode, code selector to ELD code section (D bit = 0)

`extdssel`

In Protected Mode, data selector to ELD code section (writable)

`extseg_size`

Size of the ELD code section (Bytes), para-aligned, 0 to 65_520

`extseg_used`

Amount of Bytes currently allocated to ELD instances, para-aligned, must match `eldiEndCode` of the last ELD instance

The ELD data area is referenced with the following variables. It always lives in the data/entry/stack section of the debugger.

`extdata`

Offset of ELD data area start, para-aligned. This must be at least 256.

`extdata_size`

Size of the ELD data area (Bytes), para-aligned

`extdata_used`

Amount of Bytes of the ELD data area that are currently allocated to ELD instances, para-aligned

The `extdata_size` and `extdata_used` values have to be added to the offset base in `extdata` to receive offsets in the debugger data section. The `extdata + extdata_size` calculation must not overflow and may result in up to 65_520. This maximum is reached if the `/Y=MAX` switch to `init` was specified.

16.6.2 ELD command handler

The ELD command handler allows resident ELDs to be called whenever the `cmd3` command loop of the debugger has received a command and is about to execute it. The ELD command handler is called soon after the command loop has received a command from the `getline` function. It is valid for the ELD command handler to modify the buffered text, pass on control to the subsequent command handler without changes, or completely take over the command. In the latter case, the ELD should branch back to `cmd3` once it is done executing the command.

Multiple ELDs can install command handlers. The command handlers consist of a certain structure which contains either an extcall to the label `cmd3_not_ext` (if no further ELD has installed a command handler) or a downlink made of a `rel16` near jump to the next ELD's command handler. Branching to this downlink or extcall structure allows to pass on a command (modified or not) to the next ELD, or finally to the debugger's normal command processing.

The command handler entrypoint is called with the following registers:

```
; INP:  al = first non-blank byte of command text
;       si -> subsequent command text
;       command text is stored in line_in variable
;       sp = word [saveesp]
; STT:  ds = es = ss
;       UP, EI
;       may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
```

The same protocol should generally be followed for passing on a command to the next command handler or back to the debugger's command processing.

The command entrypoint must adhere to this structure for the first ELD installing a command handler:

- A strict short jump to behind the downlink structure (2 Bytes, 0EBh 08h)
- An extcall to `cmd3_not_ext`, padded to 8 Bytes size (starts with 0E8h)

If another ELD command handler is already installed, one of the two ELDs will have its command handler modified as follows:

- A strict short jump to behind the downlink structure (2 Bytes, 0EBh 08h)
- A near jump to the next command handler (starts with 0E9h)
- Padding to 8 Bytes size

The top-most ELD command handler is pointed to by the word variable `ext_command_handler`. To install or uninstall a command handler, an ELD should use a data link to this variable.

16.6.2.1 Procedure for installing ELD command handler

The suggested procedure for installing a command handler is:

1. Adjust sizes of ELD code instance and ELD data block to installed sizes
2. Obtain value of the debugger variable `ext_command_handler`

3. If value is zero, skip to step 5
4. If value is nonzero:
 1. Overwrite first byte of our ELD's extcall with 0E9h to create a downlink
 2. Calculate displacement to write to our downlink to address the value obtained
 3. Write the displacement to our structure to finish the downlink
5. Mark our ELD as resident in the ELD instance flags, if it isn't yet
6. Write our ELD's command handler address into the debugger variable `ext_command_handler`

16.6.2.2 Procedure for uninstalling ELD command handler

The suggested procedure for uninstalling a command handler is:

1. Obtain value of the debugger variable `ext_command_handler`
2. Verify the value is nonzero, or stop the attempt as this is an error condition
3. If value matches our command handler:
 1. Check whether our handler does have a downlink structure (ie it has opcode 0E9h)
 2. If yes, calculate the offset referenced by the downlink's displacement and write this offset to the debugger variable `ext_command_handler`
 3. If no, write a zero to the debugger variable `ext_command_handler`
 4. Done
4. Else:
 1. Verify that the current handler has a downlink structure (ie it has opcode 0E9h), or stop on error condition
 2. Remember the current handler in case its downlink matches our handler
 3. Calculate the offset of the next handler from the downlink's displacement
 4. If the offset doesn't match ours yet go back to step 1
5. Copy our downlink or extcall structure to the remembered previous handler, adjusting the downlink or extcall displacement (same position) to make it work at the offset of the remembered previous handler (thus add our base to the displacement then subtract their base)
6. Done

16.6.3 ELD command injection

The ELD command injection allows an ELD to run most of the `cmd3` command loop and then regain the control flow. At that point, the ELD may either return to the `cmd3` command loop, or it may inject a command of its own creation, or it may continue the last part of the command loop which calls `getline`.

Installing command injection is done by writing an offset of a handler within the ELD code section into the debugger variable `ext_inject_handler`. Note that this variable is always cleared to zero when it is read by the `cmd3` command loop. To do command injection again, the inject handler needs to write to the variable again.

An inject handler being installed may preserve the prior value of the `ext_inject_handler` variable and restore the value upon uninstalling itself.

The inject handler is called with this protocol:

```
; INP:  sp = word [savesp]
;       line_out -> prompt message
;       di -> behind prompt message
; STT:  ds = es = ss
;       UP, EI
;       may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
```

The inject handler may usually branch to one of three entrypoints:

- `cmd3` to restart command loop
- `cmd3_not_inject` to have command loop continue to call `getline00` next (must pass `di -> behind prompt message`)
- `cmd3_injected` to have command loop accept an injected command

In the latter case, the entrypoint is to be branched to with the following protocol:

```
; INP:  al = first non-blank byte of command text
;       si -> subsequent command text
;       command text is stored in line_in variable
;       sp = word [savesp]
; STT:  ds = es = ss
;       UP, EI
;       may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
```

16.6.4 ELD preprocess handler

The ELD preprocess handler allows resident ELDs to be called whenever the `cmd3` command loop of the debugger has received a command and is about to execute it. The ELD preprocess handler is called directly after the command loop has received a command from the `getline` function (or from an inject handler). It is valid for the ELD preprocess handler to modify the buffered text, or pass on control to the subsequent handler without changes.

Preprocess handlers should not completely take over commands passed to them. The purpose of preprocess handlers is to see commands first, before they are passed to the ELD command handler chain. That means all installed preprocess handlers will see a command, even if one of the ELD command handlers will take over the command execution.

Preprocess handlers have the same structure as ELD command handlers, except they use the debugger variable `ext_preprocess_handler` to point to the first preprocess handler, and the last preprocess handler should chain to `cmd3_preprocessed`. Their installation and uninstallation procedures are the same except for using the other variable and branch destination. The protocol comment shown for ELD command handlers also applies to preprocess handlers.

16.6.5 ELD AMIS handler

This handler hooks into the debugger's AMIS interface. The handler is called early, after matching the AMIS multiplex number but before any of the implementations of debugger functions.

AMIS handlers have a similar downlink structure in their entrypoint as command handlers and preprocess handlers. They start with a strict short jump, but the rel8 displacement is only equal to 3. A downlink is composed of a strict near jump. The final AMIS handler contains a far return instruction instead, and the downlink field is padded to 3 bytes using two NOP instructions.

The handler is called with this protocol:

```
; INP:  NC
;       ds => entry segment
;       ss:sp -> far return address, ds, dx, cx, bx, ax, iret frame
;       cx destroyed
;       ax, bx, dx, bp, si, di, es, ss = original
;       fs, gs, high words = original
; OUT:  stack frame modified if desired
;       CY to iret after popping frame,
;       should retf (not use downlink)
;       NC to process function as usual (stack al = function),
;       may use downlink
; STT:  Real/Virtual 86 Mode
;       ss != debugger data/entry/stack segment
;       UP
;       DI
```

16.6.6 ELD multi-purpose puts handler

The ELD multi-purpose puts handler provides a hook into the debugger's debug terminal output. This allows an ELD to filter, store, and/or suppress output generated by other parts of the debugger or other ELDs.

A multi-purpose puts handler is installed by writing an offset in the ELD code section to the debugger variable `ext_puts_handler`. If this variable is nonzero, calls to puts (that is, all normal debugger interface output) will transfer control to the specified handler. The puts handlers now form a chain similar to command handlers, except that the variable `ext_puts_handler` is used and the final handler chains back to `puts_ext_done`. (Prior revisions of the ELDs did not create a chain of handlers.)

If to suppress the message output, the handler should resume the normal control flow of the debugger by branching to `puts_ext_done` using an `extcall` (*not* an `extcallcall`!) with CY set. The es register should be unchanged but all of ax, bx, cx, and dx need not be preserved. 386-specific register upper words should be preserved. If to continue to output the message, the handler should chain to the next handler with NC and es unchanged, and es:dx -> the message to display, length ax. The bx and cx registers never have to be preserved.

Each handler is called with this protocol:

```
; INP:  es:dx -> message to display
;       ax = length of message
```

```

;      NC
; OUT:  CY to not pass on the message for display or write to silent buff
;      have to transfer control directly to puts_ext_done (using extcal
;      ax and dx may be changed
;      NC to pass on the message,
;      may chain to next handler (or transfer to puts_ext_done)
; CHG:  bx, cx, (upon transfer to puts_ext_done with CY: ax, dx)
; STT:  ds = ss
;      UP, EI
;      may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
;      in Protected Mode, es should have a selector that
;      references a segment base which matches an 86 Mode segment

```

16.6.6.1 ELD multi-purpose puts handler: puts_ext_next entrypoint

If a multi-purpose puts handler wishes to display text that differs from what it was passed, it should call to the puts_ext_next handler. The call may be an extcall or extcallcall. The handler must pass an appropriate entrypoint into the ELD code section in cx. This is usually the chain entry of the handler, which is either a jump to a downlink or an extcall (not extcallcall) to puts_ext_done. (If the chain entry is passed, there is no risk of re-entering the calling handler.)

The protocol for puts_ext_next is as follows:

```

; INP:  es:dx -> message to display
;      ax = length of message
;      NC
; CHG:  ax, bx, cx, dx
; OUT:  -
; STT:  ds = ss
;      UP, EI
;      may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
;      in Protected Mode, es should have a selector that
;      references a segment base which matches an 86 Mode segment
;      (this is true of the debugger stack selector)

```

Passing a CY to puts_ext_next could indicate to skip the message displayed. However, if the chain entry passed in cx has a downlink to another multi-purpose puts handler then the Carry Flag is most likely ignored.

If the entry is a final transfer to puts_ext_done, the debugger function transfer_ext_cx used to bounce the control flow back into the ELD code section will now preserve the Carry Flag. (A prior bug would force NC in this transfer function for lDebugX, the _PM=1 build.)

Finally, there is no supported use for passing CY here as the effect of not displaying a message at all can be achieved by not calling puts_ext_next to begin with.

16.6.7 ELD puts copyoutput handler

This handler was added to support the co.eld (Copy Output ELD). It is called whenever the debugger actually outputs text to the debugger terminal. This does not include output written

only to the silent buffer or suppressed in the multi-purpose puts handler. Also not included are paging prompts as they are inserted later, after the puts copyoutput handler has returned.

The handlers for this hook form a chain similar to the ELD command handler. However, they use the variable `ext_puts_copyoutput_handler` and the last handler transfers control to `puts_copyoutput_ext_done`.

```
; INP:  es:dx -> message to display
;       ax = length of message
;       NC
;       byte [in_getinput] = boolean flag indicating if output
;       is from getinput session, either 0 (false) or 0FFh (true)
; CHG:  bx, cx
; OUT:  NC
;       (could set CY and transfer directly to puts_copyoutput_ext_done
;       in order to suppress output)
; STT:  ds = ss
;       UP, EI
;       may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
;       in Protected Mode, es should have a selector that
;       references a segment base which matches an 86 Mode segment
```

As noted above, a puts copyoutput handler can suppress the output passed to it. However, this is not recommended and no handler currently does this.

16.6.8 ELD puts getline handler

This handler was added to support the `co.eld` (Copy Output ELD). It is called after `getline` has received a complete input line either from `int 21h` service `0Ah` or from the `getinput` function. It is not called if a "file" type input source has yielded a line of input. This happens to match when the debugger enters a line into its line editor history, which the debugger does directly after this handler chain returns.

The handlers for this hook form a chain similar to the ELD command handler. However, they use the variable `ext_puts_getline_handler` and the last handler transfers control to `puts_getline_ext_done`.

```
; INP:  byte [line_in + 1] = length of text (excluding CR), 0 to 254
;       es:line_in + 2 -> text received, terminated by a CR
;       bl = dispatch type,
;       = 0 if input is from int 21h service 0Ah
;       = 1 if input is from getinput function
;       (note that output corresponding to this input has already
;       been written to the ELD puts copyoutput handler)
;       >= 2 if unknown dispatch type, should just pass along
; CHG:  ax, bh, cx, dx, si, di
; OUT:  pass along bl to next handler
; STT:  es = ds = ss
;       UP, EI
;       may be in Protected Mode, Real 86 Mode, or Virtual 86 Mode
;       in Protected Mode, es has a selector (stack selector) that
;       references a segment base which matches an 86 Mode segment
```

16.6.9 ELD variables

ELD variables allow an ELD to add variables to the debugger's 'isvariable?' function, which is used in the expression evaluator as well as in the R variable commands.

A certain number of ELD variables are allocated space in the list of multi-byte-text 'isvariable?' structures. The build option to add these defaults to reserving space for 16 ELD variables.

ELD variables can be used for special purpose read-only variables, as the ELD is called for the "special set up" implementation for the variable. However, writable variables are possible using this interface only if they are trivial.

An ELD should use the following to install an ELD variable:

- Data link `ext_var` points to the 10-byte ELD variable structures
- Data link `ext_var_amount` specifies how many structures exist
- Data link `ext_var_format` indicates the format of the structures, currently only format 1 is defined (this should be checked)
- Data link `ext_var_size` indicates the `ISVARIABLESTRUC_size` (this should be checked)
- Data link `isvariable_morebyte_nameheaders_ext` points to the 2-byte name headers for each ELD variable
- Data link `var_ext_setup` (albeit actually pointing at code) is provided to fill in the ELD variable structure, specifically the `ivSetup` field which must point to a near function in the `expr.asm` code section (hence an ELD has to use this particular function to branch to the ELD's code)
- Code link `var_ext_setup_done` which the ELD variable set up code should branch to using an `extcall` (*not* an `extcallcall!`) once it is done

The following protocol specifies what the variable set up code is called as:

```
; INP:  ax = array index (0-based)
;       cx = offset of this handler (ip)
;       dil = default size of variable (1..4)
;       dih = length of variable name
; CHG:  si, ax, dx
; OUT:  NC if valid,
;       bx -> var, di = 0 or di -> mask
;       cl = size of variable (1..4)
;       ch = length of variable name
```

An ELD variable should be installed by scanning the structures pointed to by `ext_var` for a structure with the first word (`ivName`) equal to zero. If this is found, the ELD variable structure is to be copied into this slot of the `isvariable?` structure array. The appropriate slot in the name headers also has to be filled with the first two text bytes of the variable name.

An ELD variable is uninstalled by clearing the first three words of the structure (`ivName`, `ivFlags`, and importantly `ivAddress`), and also clearing the name headers slot to a zero

value.

16.6.10 ELD near transfer interface

This hook allows to enter an ELD from a near call in the (first) debugger code section. The ELD can return to the near caller, too.

The entrypoint is `near_transfer_ext_entry`. This entrypoint's offset address is available as a data link in order to allow it to be entered into callback variables.

When this entry is run, the original value of `cx` is pushed, then the value of word `[near_transfer_ext_address]` is loaded to `cx`, and then control is transferred to the ELD via `transfer_ext_cx`.

Note that there is only one entrypoint and one variable for this interface. That means either it can only be used for one particular callback, or the called ELD function has to dispatch based on the near return address on the stack.

The bootloaded directory scanner, accessed by calling `scan_dir_aux`, provides several offsets in its code as data links to facilitate such dispatching:

```
..@boot_scan_dir_return_fat16_root_entry
```

Return when called as a FAT12 or FAT16 root directory entry is to be scanned, calling word `[handle_scan_dir_entry]`

```
..@boot_scan_dir_return_subdir_or_fat32_entry
```

Return when called as a subdirectory or FAT32 root directory entry is to be scanned, calling word `[handle_scan_dir_entry]`

```
..@boot_scan_dir_return_filenotfound
```

Return when called as the directory search has ended after the scan function returned CY, calling word `[handle_scan_dir_not_found]`

To return to the near caller, the ELD function should transfer the control flow to `near_transfer_ext_return` with an `extcall`, *not* an `extcallcall`. This handler will pop `cx` four times then return near.

Section 17: Command help

17.1 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)
/IN	discard command line buffer, do not run config
/A=MAX	expand auxiliary buffer to maximum, #24_576 Bytes
/A=MIN	restrict auxiliary buffer to minimum, #8_208 Bytes
/A=number	set auxiliary buffer size to hex number of bytes
/A=#number	set auxiliary buffer size to decimal number of bytes
/A	alias for /A=MAX
/X=[MAX MIN number]	change ELD code buffer size, 0 to #65_520 Bytes
/Y=[MAX MIN number]	change ELD data buffer size, 0 to #29_504 Bytes
/H=[MAX MIN number]	change history buffer size, #260 to #65_520 Bytes
/B	run a breakpoint within initialisation
/P[+ -]	append ext to initial filename and search path
/F[+ -]	always treat executable file as a flat binary
/E[+ -]	for flat binaries set up Stack Segment != PSP
/V[+ -]	enable/disable video screen swapping
/2[+ -]	enable/disable use alternate video adapter for output
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.

17.2 INSTSECT help

INSTSECT: Install boot sectors. 2018 by E. C. Masloch

Usage of the works is permitted provided that this instrument is retained with the works, so that any entity that uses the works is notified of this instrument.

DISCLAIMER: THE WORKS ARE WITHOUT WARRANTY.

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
/P KEEP	keep default/current part info handling (default)
/P AUTO	patch boot loader to use auto part info handling
/P NONE	patch boot loader to use fixed part info
/Q KEEP	keep default/current query geometry handling (default)
/Q AUTO	patch boot loader to use auto query geometry handling
/Q NONE	patch boot loader to use fixed geometry
/L KEEP	keep default/current LBA handling (default)
/L AUTO	patch boot loader to use auto LBA handling
/L AUTOHDD	patch boot loader to use auto LBA (HDD-only) handling
/L NONE	patch boot loader to use only CHS
/G KEEP	keep all current geometry (default)
/G AUTO	read all auto geometry from DOS
/G HEADS=x	set geometry CHS heads (x = KEEP, AUTO, numeric)
/G SECTORS=x	set geometry CHS sectors (x = KEEP, AUTO, numeric)
/G HIDDEN=x	set geometry hidden (x = KEEP, AUTO, numeric)
/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 18: Online help pages

18.1 ? - Main online help

```
lDebug (YYYY-MM-DD) help screen
assemble      A [address]
attach process ATTACH psp
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]
dump text table DT [T] [number]
enter          E [address [list]]
run extension  EXT [partition/][extensionfile] [parameters]
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]programe.ext [parameters]]
```

```

set command      K [[drive:][path]programe.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]
toggle 386 regs  RX
search           S range [REVERSE] [SILENT number] [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
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 pages    ?OPTIONS
Options          ?O
Boot loading     ?BOOT
lDebug build     ?BUILD
lDebug build     ?B
lDebug sources   ?SOURCE
lDebug license   ?L

```

18.2 ?R - Registers

Available 16-bit registers:

```

AX      Accumulator
BX      Base register
CX      Counter
DX      Data register
SP      Stack pointer
BP      Base pointer
SI      Source index

```

Available 32-bit registers: (386+)

```

EAX
EBX
ECX
EDX
ESP
EBP
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.

18.3 ?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.

18.4 ?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)

jno	jno	OF
jc jnb jnae	jnc jnb jae	CF
jz je	jnz jne	ZF
jbe jna	jnbe ja	ZF CF
js	jns	SF
jp jpe	jnp jpo	PF
j1 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.

18.5 ?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 operator 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.

18.6 ?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.

18.7 ?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 the SILENT keyword, 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.

18.8 ?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 8 KiB sized by default, though the /A switch can be specified to init to grow it up to 24 KiB.)

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.

18.9 ?OPTIONS - Options pages

Enter one of the following commands to get a corresponding help page:

- ?O1 DCO1 - Options
- ?O2 DCO2 - More Options
- ?O3 DCO3 - More Options
- ?O4 DCO4 - Interrupt Hooking Options
- ?O6 DCO6 - More Options
- ?OI DIF - Internal Flags
- ?OA DAO - Assembler/Disassembler Options

18.10 ?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-dependent 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 getinput function 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
- 0001_0000 gg: do not skip a breakpoint (bb or gg)
- 0002_0000 gg: do not auto-repeat
- 0004_0000 T/TP/P: do not skip a (bb) breakpoint
- 0008_0000 gg: do not auto-repeat after bb hit
- 0010_0000 T/TP/P: do not auto-repeat after bb hit
- 0020_0000 gg: do not auto-repeat after unexpectedinterrupt
- 0040_0000 T/TP/P: do not auto-repeat after unexpectedinterrupt
- 0080_0000 S: do not dump data after matches
- 1000_0000 R: do not repeat disassembly
- 2000_0000 R: do not show memory reference in disassembly
- 4000_0000 quiet command line buffer input
- 8000_0000 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 getinput function 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
- 01_0000 N: operate in MS Debug style instead of K command alike
- 02_0000 N: capitalise command line tail
- 04_0000 explicit 0-length ranges operate in partial MS Debug style
- 08_0000 R: 16-bit 80-column register dump in MS Debug style
- 10_0000 R: do variable prompts in MS Debug style
- 20_0000 R: do variable prompts with underscore separator
- 40_0000 display linebreak before R command register dump

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
- 01_0000 R: highlight changed digits (needs ANSI for DOS output)
- 02_0000 R: highlight escape sequences to int 10h, else video attributes
- 04_0000 R: highlight changed registers (overrides 01_0000)
- 08_0000 R: include highlighting of EIP
- 10_0000 set PM ss B bit
- 20_0000 break on entering Protected Mode
- 0100_0000 highlight prefix/suffix in getinput if text parts are not visible
- 0200_0000 do not call int 2F.1680 for idling
- 0400_0000 delay for a tick before writing breakpoints

- 0800_0000 do not call other IDebug instance's AMIS services
- 1000_0000 disable auto-repeat
- 2000_0000 check int 16h buffer for Control-C if inputting from int 16h
- 4000_0000 call DOS service 0Bh to check for Control-C
- 8000_0000 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
- 0010 enable 86 Mode fault interrupt hooks
- 0001_0000 force serial interrupt unhooking
- 0002_0000 force interrupt 2Fh unhooking
- 0004_0000 force interrupt 8 unhooking
- 0008_0000 force interrupt 2Dh unhooking
- 0010_0000 force interrupt 0Dh unhooking
- 0020_0000 force interrupt 0Ch unhooking
- 0100_0000 force interrupt 0 unhooking
- 0200_0000 force interrupt 1 unhooking
- 0400_0000 force interrupt 3 unhooking
- 0800_0000 force interrupt 6 unhooking
- 1000_0000 force interrupt 18h unhooking
- 2000_0000 force interrupt 19h unhooking

More options: (read/write DCO6, read DCS6)

- 0001 enable video screen swapping
- 0002 keep video screen when disabling swapping
- 0010 read key from interrupt 16h when swapping (V command)
- 0100 enable debug mode (and BU command)
- 0200 use ROM-BIOS output even when DOS available
- 0400 load and write .EXE and .COM files like flat .BIN files (/F+)
- 0800 for loading flat .BIN files set up Stack Segment != PSP (/E+)

- 1000 enable 40-column friendly mode
- 2000 in 40-column mode indent odd D lines more
- 4000 in 40-column mode display dashes at half of D length
- 01_0000 allow to share serial IRQ handler
- 0100_0000 use ROM-BIOS I/O even when DOS available (disables script file read)
- 2000_0000 display flags in style 2 for R command register dump
- 4000_0000 display flags in style 3 for R command register dump
- 8000_0000 linebreak before R register dump if not column 0 (int 10h only)

Internal flags: (read DIF)

- 00_0001 Int25/Int26 packet method available
- 00_0002 Int21.7305 packet method available
- 00_0004 VDD registered and usable
- 00_0008 internal flag for paged output
- 00_0010 DEBUG's input isn't StdIn
- 00_0020 DEBUG's input is a file
- 00_0040 DEBUG's output isn't StdOut
- 00_0080 DEBUG's output is a file
- 00_1000 state of debuggee's A20
- 00_2000 state of debugger's A20 (not implemented: same as previous)
- 00_4000 debugger booted independent of a DOS
- 00_8000 CPU is at least a 386 (32-bit CPU)
- 01_0000 internal flag for tab output processing
- 02_0000 running inside NTVDM
- 10_0000 internal flag for paged output
- 40_0000 in TSR mode (detached debugger process)
- 0100_0000 running inside dosemu
- 0400_0000 T/TP/P: while condition specified
- 0800_0000 TP: P specified (proceed past string ops)
- 1000_0000 T/TP/P: silent mode (SILENT specified)
- 2000_0000 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)
- 0100 Disassembler: 40-column friendly mode (only 4 bytes machine code per line)
- 0200 Disassembler: do not indent disassembly operands
- 0400 Disassembler: MS Debug style opcode field width
- 1000 Disassembler: access data in a16 referenced memory operand
- 2000 Disassembler: access data in a32 referenced memory operand
- 4000 Disassembler: simulate repeated a16 scas/cmps string operation
- 8000 Disassembler: simulate repeated a32 scas/cmps string operation
- 01_0000 Disassembler: hide needed MODRM keywords
- 02_0000 Disassembler: use LOOP rel, (E)CX rather than LOOPW/LOOPD
- 04_0000 Disassembler: always display MODRM keyword even if not needed

18.11 ?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 IDOS, RxDOS.2, RxDOS.3 protocols
- files' directory entries are loaded to 500h and 520h

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

- 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
- DRDOS 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
- CMDLINE pass a kernel command line (recent FreeDOS extension)

18.12 ?BUILD - IDebug build (only revisions)

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

18.13 ?B - IDebug 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

18.14 ?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

18.15 ?SOURCE - IDebug source reference

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

Releases of IDebug 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`

18.16 ?L - IDebug license

IDebug - libre 86-DOS debugger

- Copyright (C) 1995-2003 Paul Vojta
- Copyright (C) 2008-2024 E. C. Masloch

Usage of the works is permitted provided that this instrument is retained with the works, so that any entity that uses the works is notified of this instrument.

DISCLAIMER: THE WORKS ARE WITHOUT WARRANTY.

All contributions by Paul Vojta or E. 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 19: Comparison of IDebug to MS-DOS Debug

This section lists some benefits of IDebug, as compared to MSDebug. It originates in the MSDebug manual. MSDebug is based on the Debug of the 2018 free software release of MS-DOS version 2.

First, there are some differences between MSDebug and the original MS-DOS Debug which make MSDebug more similar to IDebug:

- K command to modify only the internal buffers for the program load filename, the PSP command line tail, and the PSP FCBs
- Child process allocated using interrupt 21h function 48h and initialised using function 55h
- After termination of debuggee a new child process is created
- Restores interrupt 1 and 3 vectors (similar to IDDebug)
- BU command like IDDebug's
- .HEX file read ends on NUL byte (0), EOF byte (1Ah), or End Of File
- Bugfix: Move command M will correctly move forwards or backwards based on the linear address, fixing overlapping moves in which the segments differ in the opposite way to the linear addresses

Here's the advantages of IDebug as compared to MSDebug:

- Expression evaluator can be used wherever a number is to be parsed
- D, U, T, TP, P, G commands can be repeated with autorepeat
- Permanent breakpoints (B commands)
- G command can re-use prior breakpoints list with G AGAIN command (or autorepeat)
- Several variables beyond the 16-bit registers to control the debugger or store calculation results
- Paging to allow reading longer outputs
- 686 level assembler and disassembler
- DPMI build IDebugX for debugging DPMI clients, supporting 32-bit offsets in segments
- InDOS mode to switch to ROM-BIOS video and keyboard I/O rather than DOS's standard output and input

- Line editor and line history (if not using the DOS line input function)
- Can be boot loaded for debugging Real/Virtual 86 Mode kernels
- Can be installed as a device driver in CONFIG.SYS
- Script for IDebug file reading using the Y command
- Serial I/O mode
- Conditional tracing for the T, TP, and P commands using a condition after a WHILE keyword
- Buffered tracing for T, TP, and P using a SILENT keyword
- T defaults to proceeding past software interrupt calls, can be modified using Trace Mode (TM command)
- TP command which proceeds past repeated string instructions
- DM command to list MCBs
- DI command to dump interrupt handler chains
- DW and DD commands to dump data in words or dwords
- 32-bit numeric handling
- F and S commands can use a memory source instead of a list, using a RANGE keyword
- H command can display an expression result in hexadecimal, decimal, or one arbitrary base of choice
- I and O commands have IW/OW and ID/OD variants for word and doubleword port I/O
- R command can be switched to display 32-bit and 386 registers
- Machine type can be set to make the assembler and disassembler display when the machine does not support an instruction
- QA command can be used to try to terminate an attached process
- S command can search backwards using the REVERSE keyword
- Provides a number of online help pages
- RN command to dump 8087 registers
- RM command to dump MMX registers, and variables to read and write them
- RE command buffer to run commands from T, TP, P, or G dump calls
- RC command buffer to run commands at startup or group several commands later on
- Numeric inputs can be specified with a hash sign ‘#’ modifier to enter in arbitrary numeric bases
- Access variables and VALUE IN constructs to detect memory accesses before they are actually carried out (requiring to trace instructions)

- TSR and ATTACH commands to detach from or attach to a process
- IF command to conditionally run another command
- Timer and AMIS interrupt hooks (optional)

Furthermore, lDebug can be built using the free software Netwide Assembler, rather than relying on a binary-only Microsoft Macro Assembler. (The assembler executable shipped with MSDebug is technically free, but it does not have sources.)

However, there are some disadvantages to lDebug as well:

- Memory use and executable size can easily reach as much as ten times that of MSDebug
- Less compatibility to original MS-DOS Debug
- Performance may be worse
- May require some MS-DOS version 3 or version 5 features

Section 20: Test Reference

This chapter lists all tests in the test suite.

20.1 **test_beep**

Runs the command `... invalid command` and checks that the returned error carat display contains a bell codepoint (byte 07h, U+0007).

20.2 **test_build**

Checks the `?build` and `?version` commands.

20.3 **test_rh**

Checks the Register dump History mode. Includes subtests for many RH commands. Refer to section 10.38.

20.4 **test_dt**

Checks the DT command text table and the DT command to dump byte values' corresponding texts. Refer to section 10.17.

20.5 **test_rr_status**

Checks that the R command's jumping notices are correct. Also checks for flag status display, referenced memory contents display, and the `[needs 386]` machine requirement display. Includes subtests for many conditions. Refer to section 10.37.

20.6 **test_aa_basic**

Tests the assembler's output. Includes many subtests, some of which are expected failures.

20.7 **test_rr_basic**

Tests the R command to display and modify variables, either debugger variables or memory indirect variables. Includes subtests for a number of different commands. Some of the tests also use complex expressions to test the expression evaluator. Some of the subtests contain no-op commands (commented out as `; nothing`) that only check an additional result of a prior subtest.

20.8 **test_misc**

Miscellaneous subtests:

- S command tests

- D command tests, including D TOP, dollar sign addresses, and pointer type expressions
- Reading variables using the R command, up to DCO7 and DIF7 (with expected failures for DCO8 and DIF8)
- Calculations using the H command and various expression evaluator constructs
- Reading debugger variables (PSP variables and registers) using the H command
- Reading some compound variables using H commands
- Checking the equality of some compound variables
- Checking value limits on assigning to AX or an indirect word variable

Further miscellaneous tests include:

- Autorepeat for T command, without or with blanks
- Disable autorepeat
- H command expression evaluation overflow
- Register change highlighting
- Line editor history
- Line editor
- Sleep command
- Paging of long command output
- C command
- T/TP/P commands
- F command with overlong pattern
- F command with a range instead of list pattern
- S command with range, too long range, too long list

20.9 test_timeout

Instructs the debugger to sleep so long that the test tear down handler will terminate the debugger process.

20.10 test_int2D_unhook

Installs the debugger's AMIS handler (int 2Dh handler) then installs a tiny handler on top of it, testing how attempting to unhook the debugger's handler works in this case. The tiny handler is first of the form '90 EA offset segment 00 00' (no IISP header detected) and then of the form '90 EA offset segment "KB"' (uninstalled iHPFS style IISP header detected).

This test requires running under a DOS.

20.11 test_bb_gg

Tests permanent (bb) and temporary (gg) breakpoints.

20.12 test_bb_fill

Tests permanent breakpoint setup, listing, and overflows of structures.

20.13 test_access_var

Checks for access variables working. Refer to section 12.19.

20.14 test_dpmimini

Enters Protected Mode using a small DPMI test case.

First, if need be a DPMI host may be run. Next, the client executable is loaded. Subsequently the debugging hint in text form is searched within the loaded program. It is checked that PM is entered successfully.

The fix for a bug in the R command is tested next, except under IDDebugX where a failure would crash the debugger. The PM interrupts hooked by IDebugX are checked, except when IDDebugX is used. Several subtests check PSP variables.

Finally, the return to 86 Mode via client process termination is checked.

This test requires running under a DOS, for a DPMI host to be available, and for an IDebugX build to be used (detected by the "x" in the build name).

20.15 test_dpmioffs

This test has a setup similar to test_dpmimini. Two different breakpoints are detected in the debugging hint texts. At both of the breakpoints, several C commands are tested. One point of the tests is to insure that 32-bit offsets in part of an expression such as memory indirection or a LINEAR expression do not set the 32-bit offset status for the entire expression's caller.

This test requires running under a DOS, for a DPMI host to be available, and for an IDebugX build to be used (detected by the "x" in the build name).

20.16 test_dpmialoc

This test has a setup similar to test_dpmimini. It checks that when a permanent breakpoint is set in a DPMI allocation (beyond 1088 KiB) and then the debugger is entered in 86 Mode via mode switch (not by terminating the client) and then an interrupt instruction is to be disassembled, the debugger will not have losts its Extra Segment and the disassembly operand stack special opcode scan will succeed.

This is based on a bugfix in the mode switching done by the debugger in this circumstance, added as a test case afterwards, both in 2022 April.

This test requires running under a DOS, for a DPMI host to be available, and for an IDebugX build to be used (detected by the "x" in the build name).

20.17 test_missing_executable

This test tries to load nonexisting executables specified to the N command followed by an L command. First a file should not be found and then a path.

This test requires running under a DOS.

20.18 test_error_executable

This test tries to load a corrupt executable specified to the N command followed by an L command. The MZ executable header specifies a too large image for any 86 Mode operating system in this executable.

This test requires running under a DOS.

20.19 test_load_boot

This checks that several BOOT commands work.

An IDOS protocol load is attempted to run without a failure, and its entrypoint address (200h:400h) is checked. A check value mismatch is forced. The RxDOS.3 protocol load is attempted with the default filename, which should not be found. A FreeDOS protocol load is attempted to run without a failure, and its entrypoint address (60h:0) is checked. A boot directory listing is attempted and the presence of the debugger executable and startup script is matched.

The boot sector of the debugger diskette is loaded to segment 1000h and it is checked that the informational 'FAT12' identifier shows up exactly once in the sector. The 55h, AAh signature is checked in the sector. The boot sector is read repeatedly to segment 1020h with implied start sector and length, explicit start sector but implied length, and all explicit parameters. The sectors are matched with the first read. Also, it is insured that a sentinel byte after the 512 Bytes for the subsequent reads does not change.

This test requires running without a DOS.

20.20 test_yy

Tests Y commands to read Script for lDebug files.

A simple script is tested first. Then 20 nested scripts are tested, expecting an error that the nesting is too deep but still executing all commands from the opened files in the correct order. Then 3 nested scripts are tested, which should not cause an error.

Subsequently, sleeping within nested scripts is tested. Further, cancelling a running command with Control-C (codepoint U+0003) is tested, utilising the sleeping scripts. The first cancellation is done with IOL equal to zero, to cancel only the sleep command and none of the Scripts for lDebug. (Refer to section 12.9.6.) The second involves IOL equal to one, cancelling the sleep command as well as one script level. The third involves IOL equal to one again but with two Control-C codes sent from the terminal, twice cancelling one sleep command and one script level. The fourth cancellation uses IOL equal to two, cancelling the sleep and two script levels.

A further SLD tests calling subfunctions (labels) within the same SLD file. A final SLD tests visibility of commands in a Script for lDebug using the '@' prefix as well as setting YSF flag 4000h to hide some of the commands. (Refer to section 12.15.1.)

20.21 test_double_ctrlc

This test sends a double Control-C from the serial terminal to the debugger. At this point the debuggee is running in an idle loop so that the double Control-C can act as a breakpoint. In order to work, the timer interrupt 8 is hooked by the debugger.

After the debugger breaks out of the loop, the code segment is matched against the debuggee code segment. If it matches and the offset is plausible, the test is considered finished. Otherwise, the debugger attempts to trace-proceed out of the current interrupt handler until it has executed an `iret` or `retf imm16` instruction and checks the segment for the debuggee code segment after.

20.22 test_eee_interactive

This tests the interactive enter mode (section 10.18).

20.23 test_rc

This tests the RC (command line) buffer and execution from it. First a single R command is written and ran. Then two more R commands are appended and the RC buffer is ran again.

Then a small loop is written into the RC buffer using IF and GOTO commands, as well as labels. The display of the commands themselves is disabled by using '@' prefixes. The correct amount of iterations as well as hiding of commands is tested.

A longer loop is ran next, but the loop is placed into the RE buffer instead, with the RC buffer containing only an 'RE' command. The RCLIMIT variable (section 12.6.3) is set so low that it would abort the loop if the RE buffer commands were to be counted against it.

A similar construct is used next, except that the loop is in a Script for IDebug file and the RC buffer contains a Y command to load this SLD.

20.24 test_ext_extlib

Runs the help of the extlib Extension for IDebug. Both on its own, with the long descriptions, and with the wide list of ELDs. It is checked that at least 40 ELDs are listed.

20.25 test_ext_ldmem

Runs the ldmem Extension for IDebug. The MEM and ELD keywords are tested by directly running the ELD. The ELD noun should display the transient ELD.

Then the ELD is installed residently. The ELD noun is used first. It is checked that the resident ELD is displayed. Next, COMMANDHANDLER, INJECTHANDLER, STACK, and HISTORY nouns are ran. Finally the resident ELD is uninstalled.

20.26 test_ext_aformat

Installs the aformat Extension for IDebug then tests it by assembling several instructions.

20.27 test_ext_checksum

Installs the checksum Extension for IDebug and runs it on several data blocks. The results are checked.

20.28 test_ext_list

Runs the transient list Extension for IDebug. The first run is with an empty command line tail. As only the extlib.eld is known to be available it is specified as the ELD file to list for the second run. Contents and format of the output are tested.

20.29 test_ext_amitsrs

This runs the Extension for IDebug that lists multiplexers installed according to the Alternate Multiplex Interrupt Specification (AMIS). To make sure at least one multiplexer is installed, this test runs 'install amis' initially, which installs the debugger's AMIS handler.

The amitsrs command is run as is, and then with the keywords 'int', 'mpx', and 'verbose'. In the interrupts display it is insured that at least one multiplexer hooks interrupt 2Dh (necessary) and interrupt 3 (always true for normal IDebug builds).

20.30 test_ext_reclaim

In this test, a resident ldmem Extension for IDebug is installed. It is then tested that running the help for extlib.eld leaves a transient ELD installed. Finally, the reclaim ELD is used to reclaim this transient ELD. (Reclamation is built-in to the debugger's ELD loader as well, however this will not reclaim the last loaded ELD when it returns control to the debugger.)

20.31 test_ext_amount

This test installs the amount Extension for IDebug, which provides the ELDAMOUNT variable. It is checked that this variable reads as at least 2, one each for the LINKCALL ELD and the amount ELD. Then an ldmem.eld is installed and it is checked that the value in the ELDAMOUNT variable increments.

20.32 test_ext_alias

Tests the alias Extension for IDebug. Installs the ELD, then runs a list command, adds an alias, lists the alias, runs the alias, and deletes the alias again.

20.33 test_ext_dosseek

To test the dosseek Extension for IDebug, a small stub is assembled first to open a file handle. The extlib.eld file is opened, as it always exists. Then the dosseek.eld is used to get the current seek, set it to 4096, get it again, and then set it to the EOF. The final seek is checked to match the filesize. Finally, it is attempted to get the seek for file handle #19 (valid process handle but closed) and FFFF (never a valid process handle) and the error messages are checked.

This test requires running under a DOS.

20.34 test_ext_history

Installs the history Extension for IDebug, and checks that its show and clear commands work.

20.35 test_ext_amismsg

Installs the debugger's AMIS handler and the amismsg Extension for IDebug. Several small handlers that pass messages to AMIS function 40h are assembled. An overflow is tested to

truncate a message that is too long, along with returning a different status in AL. AMIS function 41h is tested as well.

20.36 test_ext_amiscmd

Installs the debugger's AMIS handler and the amiscmd Extension for lDebug. An rvm command is injected using AMIS function 43h. Next the same command is injected again but with all flags in CX set, which should be rejected. After this, four commands are injected at once. Then 16 commands are injected at once, padded to the maximum length with blanks. This should overflow the ELD's buffer. Finally two differing commands are injected and it is insured that the order of injection matches the order of the calls into the ELD.

20.37 test_ext_amisoth

Installs the debugger's AMIS handler and the amisoth Extension for lDebug. The AMIS function 42h is tested. The status returned in AL is insured to be FFh. The segment returned in DX is matched to equal the Debugger PSP. The ELD link info header (refer to section 16.3) is tested a little. The signature is matched as E1D1h, the use link hash word is matched as 1 or 0, and the amount of data and code links are both matched to be at least 64 and at most 256.

Section 21: 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.

One of the Extensions for lDebug, dbitmap.eld, contains a font copied from the GLaBIOS project. Its license follows.

21.1 GLaBIOS font license (used for dbitmap.eld)

Font bitmaps from "VileR", (CC BY-SA 4.0)

<https://int10h.org/oldschool-pc-fonts/>

Copied from <https://github.com/640-KB/GLaBIOS/blob/b60b2549372e30447ffa454d4ae487390f91d509/src/GLABIOS.ASM#L10975> with the backslash comment fixed to avoid a NASM misfeature.

According to https://int10h.org/oldschool-pc-fonts/readme/#legal_stuff this font insofar as it is copyrightable is available under: Creative Commons Attribution-ShareAlike 4.0 International License.

21.2 BriefLZ depacker usage conditions

BriefLZ - small fast Lempel-Ziv

8086 Assembly IDOS 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.

21.3 LZ4 depacker usage conditions

8086 Assembly IDOS iniload payload LZ4 depacker

by E. 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.

21.4 Snappy depacker usage conditions

8086 Assembly IDOS iniload payload Snappy depacker

by E. 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.

21.5 Exomizer depacker usage conditions

8086 Assembly IDOS iniload payload exomizer raw depacker

by E. 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.

21.6 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.

21.7 Heatshrink depacker usage conditions

8086 Assembly IDOS iniload payload heatshrink depacker

by E. 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.

21.8 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.

21.9 LZO depacker usage conditions

8086 Assembly IDOS iniload payload LZO depacker

by E. 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.

21.10 LZSA2 depacker usage conditions

8086 Assembly IDOS iniload payload LZSA2 depacker

by E. 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.

21.11 aPLib depacker usage conditions

8086 Assembly IDOS iniload payload aPLib depacker

by E. 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.

21.12 bzpack depacker usage conditions

8086 Assembly IDOS iniload payload bzpack depacker

by E. 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 86e9f2b26187, from commit on at 2024-09-15 15:39:15 +0200

If this is in ecm's repository, you can find it at
<https://hg.pushbx.org/ecm/ldebug/rev/86e9f2b26187>