Quick and Dirty DEBUG Tutorial.

Note: MS-Debug means the original Debug written by Microsoft.
DEBUG/X means the MS-Debug clone (both versions).
DEBUG means both the original and the clone.
Enhancements for DEBUG/X are put in square brackets: [ENH:].

MS-Debug was developed originally by Tim Paterson as a tool to aid in his development of the 16-bit x86 OS that eventually became MS-DOS Version 1.0. A version of MS-Debug was included with every version of MS-DOS (and PC-DOS, Compaq DOS, etc) and with every version of Windows through Windows XP.

DEBUG supports the 8086/8088 and 8087 registers and instruction sets.

[ENH: DEBUG/X supports the 80386 registers and instruction sets up to the Pentium Pro, including Floating Point instructions. The MMX instruction set is missing, however. It is planned - and partially implemented - for DEBUGX].

DEBUG command line

For the version of MS-Debug that is included with Windows 98 SE, the command line "DEBUG /?" returns:

Runs Debug, a program testing and editing tool.

DEBUG [[drive:][path]filename [testfile-parameters]]

[drive:][path]filename	Specifies the file you want to test.
testfile-parameters	Specifies command-line information required by the file you want to test.

After Debug starts, type ? to display a list of debugging commands.

DEBUG commands

This section deals with the commands that DEBUG recognizes while in command mode. Command mode, indicated by the presence of the DEBUG command prompt (a hyphen), is the mode that DEBUG starts in.

For the more recent versions of DEBUG, you can cause DEBUG to display a list of the commands and their parameters by entering a "?" at the DEBUG command prompt.

DEBUG recognizes only hex numbers (without a trailing "H").

DEBUG maintains a set of variables that it uses to store a target program's CPU registers. DEBUG initializes these variables each time it is started. The variables for the AX, DX, BP, SI, and DI registers are set to 0. If a target program or other file is loaded, the variables for the BX and CX registers are set to the size of the program or file. Otherwise, the variables for the BX and CX registers are set to 0. If a target program is loaded, the variables for the segment, IP, and SP registers are set to values appropriate

for the loaded program. Otherwise, the variables for the segment registers are set to the segment address of the area of memory that DEBUG uses to store the output from the Assemble command, the variable for IP is set to 100h, and the variable for SP is set to FFFEh.

[ENH: DEBUG/X supports the 32bit registers introduced with the 80386. The high-words of these registers are initialized to zero on program start.]

Note that any DEBUG command that accepts a CPU register as a parameter is actually using one of these variables. For the remainder of this section, all register names refer to the associated variable.

For the commands that accept a range parameter, the parameter can be used to specify a starting address, or a starting and ending address, or a starting address and a length. To differentiate a length parameter from an ending address, the length parameter must be preceded with an "L". The starting address parameter can include a segment register or a segment address constant, separated from the offset address with a colon. Some examples using the Dump command:

```
-D 0
-D CS:0
-D F000:FFF6
-D 0 F
-D DS:100 102
-D SS:20 L 4
```

[ENH: DEBUGX understand this additional syntax:

```
#D $F000:FFF6
```

which tells DEBUGX that F000 should be interpreted as a segment, not as a selector when debuggee is in protected-mode.

If debuggee is in protected-mode and the segment part of the starting address is a 32bit selector, the offset parts of the starting and ending address - as well as the length - can be 32bit. Examples:

```
-D DS:110000 110FFF
-D CS:400000 L 20000

]
```

The Assemble command (A) is used to enter assembly mode. In assembly mode, DEBUG prompts for each assembly language statement and converts the statement into machine code that is then stored in memory. The optional start address specifies the address at which to assemble the first instruction. The default start address is 100h. A blank line entered at the prompt causes DEBUG to exit assembly mode.

```
Syntax: A [address]
```

The Compare command (C) compares two memory regions. All differences will be listed, both addresses are displayed, with their respective data bytes. Example:

```
-C 0100 l 2 0200
1234:0100  E9  65  1234:0200
1234:0101  01  02  1234:0201
```

Syntax: C range address

The Dump command (D), when used without a parameter, causes DEBUG to display the contents of the 128-byte block of memory starting at CS:IP if a target program is loaded, or starting at CS:100h if no target program is loaded. The optional range parameter can be used to specify a starting address, or a starting and ending address, or a starting address and a length. Subsequent Dump commands without a parameter cause DEBUG to display the contents of the 128-byte block of memory following the block displayed by the previous Dump command.

[ENH: there is a autorepeat function implemented for D. If the last command entered was Dump, pressing a single ENTER will count as a subsequent Dump command.]

Syntax: D [range]

[ENH:

DEBUGX only: the Dump Interrupt (DI) command displays interrupt vectors in real/v86-mode and DPMI interrupt (+exception) vectors in protected-mode. Examples:

Display interrupt 00 in real-mode:

```
-DI 0
int 00 02F8:03F8
```

Display interrupt and exception vectors 08-0B in 32bit protected-mode:

```
#DI 8 4
int 08 00C7:00000D28 00C7:00001228
int 09 00C7:00000D2D 00C7:0000122D
int 0A 00C7:00000D32 00C7:00001232
int 0B 00C7:00000D37 00C7:00001237
```

Syntax: DI vector [count]

DEBUGX only: the Dump LDT descriptor (DL) command displays DPMI LDT descriptors. Example:

```
#dl 84 2
```

```
0084 base=00110800 limit=0000014F attr=00F2
008C base=00110000 limit=000007FF attr=00F3
```

DL can also be used to display limit and attributes of those GDT descriptors which are accessible by the client. Example:

```
#dl 18 3
0018 base=???????? limit=FFFFFFFF attr=CFFB
0020 base=???????? limit=FFFFFFFF attr=CFF3
0028 base=???????? limit=???????? attr=????
```

Syntax: DL selector [count]

The Dump MCBs (DM) displays the current DOS PSP and the DOS Memory Control Block (MCB) chain. Example:

```
PSP:1A27
0219 4D 0008 04D6 SD
06F0 4D 06F1 00BA COMMAND
07AB 4D 0000 0007
07B3 4D 06F1 0047 COMMAND
07FB 4D 0000 0039
```
...

First value is the segment address of the MCB, followed by 4D/5A, the owner and size of the MCB and finally the 8-byte name of the owner. An owner of 0000 means that this MCB describes a free memory block.

Syntax: DM

DEBUGX only: The Dump eXtended memory command (DX) allows to examine the full range of memory. It uses Interrupt 15h, AH=87h for extended memory access. It should be noted that, when running under a Expanded Memory Manager (EMM), Interrupt 15h, AH=87h behaves slightly inconsistent: addresses below 110000h are regarded as linear, while addresses >= 110000h are regarded as physical.

Syntax: DX [address]
]

The Enter command (E) is used to enter data into memory. The required address parameter specifies the starting location at which to enter the data. If the address parameter does not specify a segment, DEBUG uses DS. If the optional list parameter is not included, DEBUG displays the byte value at the specified address and prompts for a new value. The spacebar or minus key can be used to move to the next or previous byte. Pressing the Enter key without entering a value terminates the command. If the

list parameter is included, it can contain up to 8 byte values separated by spaces, commas, or tabs, and/or a case significant string enclosed in single or double quotes.

Examples:

```
-E 100 01 02 03
-E 100 'ABC'
-E 100 'ABC' 0
```

Syntax: E address [list]

The Fill command (F) fills a memory regions with a byte pattern. To fill for example the first 8000h bytes of current data segment with pattern 55 AA, enter:

```
-F 0 L 8000 55 AA
```

Syntax: F range list

The Go command (G), when used without a parameter, causes DEBUG to copy the contents of the register variables to the actual CPU registers and effectively jump to the instruction at CS:IP, giving full control to the program being debugged. DEBUG will reassert control when the program terminates normally, or DEBUG will not reassert control. The optional address parameter can be used to specify the starting address. The optional breakpoint list can be used to specify up to 10 temporary breakpoint addresses. To differentiate the address parameter from the breakpoint list, the address parameter must be preceded with an "=". Note that the breakpoint addresses must coincide with the start of a valid instruction. To resume execution after a breakpoint, use the Go command without a parameter.

Syntax: G [=address] [break0 break1...]

The Hex command (H) computes the sum and difference of two hex numbers.

Syntax: H value1 value2

[ENH: MS-Debug will handle WORD values only, DEBUG/X will accept DWORD values.]

The Input command (I) causes DEBUG to read and display the value from the 8-bit I/O port specified by the required port parameter.

Syntax: I port

[ENH: besides I there exists IW/ID to read a WORD/DWORD from a 16-bit/32-bit port. ID is available

on machines with a 80386+ cpu only.]

The Load command (L) is used to load a file into memory. The file to load is specified with the Name command (N). The optional address parameter specifies the load address. The default load address is CS:100 for all files other than EXE files, for which information in the file header determines the load address. After DEBUG loads the file it sets BX:CX to the file size in bytes. Recent versions of DEBUG display "Extended Error 1282" if the specified file cannot be found.

Syntax: L [address]

The Load command can also be used to load one or more logical sectors from a disk drive into DEBUG's memory. The drive should be specified as 0=A, 1=B, 2=C etc.

Syntax: L address drive startsector numberofsectors

[ENH: L with logical sectors on Windows XP will work for DEBUGX only.]

The Move command (M) copies a memory region to another location. If source and destination overlap, parts of the source will be overwritten.

Syntax: M range address

[ENH: if M is entered without - or with one - argument, it is regarded as 'Mode' command, which displays - or sets - the current CPU mode.

Syntax: M [mode]

'mode' is 0 for a 8086/8088, 1 for a 80186, ... 6 for a 80686 (Pentium=80586, Pentium Pro=80686).

The MC command allows to set the FPU mode. This is for the A and D commands only. To display the current FPU mode, use the M command.

Syntax: MC [N|2]

MC without argument will enable and MC N will disable the FPU. MC 2 will set the FPU to 80287 - this is accepted only if the current cpu is a 80386.
]

The Name command (N) is used to input a filename (actually, a file specification that can include a drive and a path) for a subsequent Load or Write command. Because the Name command uses the simulated PSP as a buffer, it can also be used to enter command line arguments and/or switches for the program being debugged.

Be aware that the N command alters the value of the AX register!

Syntax: N filespec [arguments]

The Output command (O) causes DEBUG to write the byte specified by the required byte parameter to the 8-bit I/O port specified by the required port parameter (O port byte)

Syntax: O port byte

[ENH: besides O there exists OW/OD to write a WORD/DWORD to a 16-bit/32-bit port. OD is available on machines with a 80386+ cpu only.]

The Proceed command (P) functions like the Trace command (T) for all instructions other than the repeat (REP) string, LOOP, procedure CALL, and interrupt call (INT) instructions. For a repeat string instruction, DEBUG executes the instruction to completion. For a LOOP instruction, DEBUG executes the entire loop to completion. For a procedure CALL or an interrupt call instruction, DEBUG executes the entire call up to and including the return instruction. After execution is complete, DEBUG continues as it does for the Trace command.

Syntax: P [=address] [number]

The Quit command (Q) should be a no-brainer.

Syntax: Q

The Register command (R), when used without a parameter, causes DEBUG to display the contents of the target program's CPU registers. The optional register parameter will cause DEBUG to display the contents of the register and prompt for a new value. When DEBUG displays the contents of the registers, it encodes the processor flags as follows:

Flag	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE	PO
Carry	CY	NC

Syntax: R [register]

[ENH: if the register argument is given, it can be immediately followed by the new value . No prompt will appear then.

Syntax: R [register] [value]

The processor flags can be changed in two ways. The first syntax changes individual flags:

-R F [OV|NV|DN|UP|EI|DI|NG|PL|ZR|NZ|AC|NA|PE|PO|CY|NC]

the other syntax will set the [E]FL register:

-R [E]FL <[d]word value>

DEBUGX only: The RM command displays MMX registers. The content is displayed from low to high bytes!

Syntax: RM

The RN command display the content of the floating point registers. DEBUGX renders the content of these registers in "readable" floating point format, DEBUG displays the content in raw hex format (because the rendering code requires some space and DEBUG is intended to be kept small in size).

Syntax: RN

The RX command is active if the current cpu is a 80386 or better. Then RX will toggle between the simple 16bit register display (which is the default) and the full 32bit register display.

Syntax: RX

]

Search (S) is used to find the occurrence of a specific byte or series of bytes within a segment. For example, to find all occurrences of 'CD 21' in memory region DS:100-1FF, enter:

```
-S 100 l 100 CD 21
 1234:0122
 1234:01EC
```

Or, to find all strings "hello" in DS:0100-FFFF, enter

```
-S 100 FFFF 'hello'
 1234:0122
 1234:1221
 1234:EADC
```

Syntax: S range list

The Trace command (T), when used without a parameter, causes DEBUG to execute the instruction at CS:IP. Before the instruction is executed, the contents of the register variables are copied to the actual CPU registers. After the instruction has executed, and updated the actual CPU registers (including IP), the contents of the actual CPU registers are copied back to the register variables and the contents of these variables are displayed. The optional address parameter can be used to specify the starting address. The optional count parameter can be used to specify the number of instructions to execute. To differentiate the address parameter from the count parameter, the address parameter must be preceded with an "=". Note that the first byte at the specified address must be the start of a valid instruction.

Syntax: T [=address] [number]

[ENH: if the current CS:IP points to an INT opcode, DEBUG/X's behaviour depends on the current value of the TM option. If it is 0 (the default), the debugger will regain control when the INT has been processed. If TM is 1, the debugger will step "into" the INT. Furthermore, autorepeat is active for T. That is, if the last command was T, a single ENTER will count as a repeated T command.]

The Unassemble command (U), when used without any parameter, causes DEBUG to read approximately 32 bytes of memory starting at the initial CS:IP values specified in the file header if an EXE file is loaded, or starting at CS:100h if any other type of file or no file is loaded. For each instruction in this memory, DEBUG disassembles (translates from machine code to assembly language) the instruction and displays the address, the machine code, and the corresponding assembly language. The optional range parameter can be used to specify a starting address, or a starting and ending address, or a starting address and a length. Subsequent Unassemble commands without any parameter start at the instruction following the last instruction processed by the previous Unassemble command.

[ENH: Autorepeat is active for U. That is, if the last command was U, a single ENTER will count as a repeated U command.]

Syntax: U [range]

The Write command (W) causes DEBUG to write BX:CX bytes to the output file. The optional address parameter specifies the starting source address. The default starting source address is 100h. Note that loading a file from the DEBUG command line or with the Load command will set BX:CX to the size of the file, a feature clearly intended to aid in the patching of executables.

Syntax: W [address]

The Write command can also be used to write one or more logical sectors to a disk drive. The drive should be specified as for the Load command.

Syntax: W address drive startsector numberofsectors

WARNING: You should know what you are doing when using this command. If used inappropriately, W can cause severe data losses.

[ENH: W with logical sectors on Windows XP will work for DEBUGX only.]

For the Trace, Proceed, and Register commands, after DEBUG completes the operations described above, it disassembles the next instruction and displays the address of the instruction, the machine code, and the corresponding assembly language. For the Trace and Proceed commands, if the instruction contains a memory operand, DEBUG displays the address that the operand refers to and the contents of that address. The segment register specified in the address is the default segment register for the instruction, or the segment register specified by a segment override.

Syntax:

One perhaps non-obvious use for the Register command would be to control the starting address for the next Trace or Proceed command by loading a new value into CS and/or IP. And along the same lines, non-obvious uses for the Assemble and Enter commands would be to alter the instructions and/or data for the program being debugged.

When DEBUG processes a Trace, Proceed, Register, or Unassemble command, it displays assembly language (as described above) for any byte value or sequence of byte values that it recognizes as a valid instruction. For any other byte value, DEBUG displays the DB pseudoinstruction followed by the byte value. DEBUG may fail to recognize a byte value as part of a valid instruction for any of the following reasons:

The byte is part of an instruction that is specific to a later generation x86 processor.

The disassembly process did not start at the first byte of a valid instruction.

The byte is data.

Note that DEBUG's ability to execute an instruction (for the Trace and Proceed commands) does not depend on its ability to recognize the instruction.

Instruction operands

The instruction operands are the objects on which the instruction "operates". For DEBUG, these objects can be constants, registers, or memory. A memory operand specifies the offset address of the data to operate on. To differentiate memory operands from constants, memory operands must be enclosed in square brackets.

For a direct memory operand, the offset address of the data is specified in the operand. For an indirect memory operand, the offset address of the data is calculated at run-time using the contents of one or two registers and an optional displacement. In 16bit code, the registers can be a base register (BX or BP) or an index register (SI or DI), or one of each. The following examples illustrate the differences between constant operands, register operands, direct memory operands, and indirect memory operands:

```
MOV   AX,100   ; Moves the value 100 into AX
MOV   AX,BX    ; Copies the value in BX to AX
MOV   AX,[100] ; Copies the word at DS:100 to AX
MOV   AL,[BP+2]; Copies the byte at DS:BP+2 to AL

JMP   100      ; Destination offset is 100
JMP   AX       ; Destination offset is the value in AX
JMP   [100]    ; Destination offset is the word at DS:100
JMP   [BX]     ; Destination offset is the word at DS:BX
JMP   [BX+4]   ; Destination offset is the word at DS:BX+4

  ; Near calls work just like the jump instructions above.
```

DEBUG statements

This section deals with the statements that DEBUG recognizes while in assembly mode.

Each statement occupies a single line. For each statement, the first character that is not a space or a tab must be a semicolon or the start of a valid instruction mnemonic. A semicolon, at the start of a statement or anywhere within a statement, causes DEBUG to ignore the remainder of the line.

As previously noted, DEBUG recognizes only hex numbers (without a trailing "H").

As previously noted, memory operands must be enclosed in square brackets.

In MS-Debug, segment override prefixes must be placed on the line preceding the prefixed instruction. For example:

```
  ES:
  MOV AX,[100]
```

[ENH: DEBUG/X rejects this format. Here exist two alternative ways:

```
  SEG ES
  MOV AX,[100]
```

or

```
  MOV AX,ES:[100]
```

]

In addition to the items detailed above, DEBUG statements can include the following:

The CPU register names (other than IP):
AX BX CX DX
AL AH BL BH CL CH DL DH
SP BP
SI DI
CS ES DS SS

[ENH: if cpu is 80386 or better, the extended registers are recognised as well: EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI.

Furthermore, there are 2 new segment registers, FS and GS.]

The words NEAR (or NE) and FAR, optionally followed by PTR, which are used to override the default forms of the CALL and JMP instructions. The default form of a JMP instruction can be short, near, or far, depending on the destination. The default form of a CALL instruction can be near or far, depending on the destination. Note that DEBUG will default to far only if the destination is recognizable as a far address, which means that it must be a constant of the form segment:offset (for example, "JMP F000:FFF0"). If the destination is specified by a memory operand (for example, "JMP [100]"), DEBUG will default to near. The optional PTR has no effect on the assembly process.

The words BYTE (or BY) and WORD (or WO) [ENH: and DWORD (or DW)], optionally followed by PTR, which are used to specify the type for memory data. For instructions that take two operands, DEBUG will assume that the type for the data specified by a memory operand matches the type for the other operand. For instructions that take one operand, the type must be specified. The optional PTR has no effect on the assembly process.

[ENH: for floating point opcodes, there exist 3 additional memory types, FLOAT (or FL), DOUBLE (or DO) and TBYTE (or TB):

```
FLD  FLOAT [100]
FSTP DOUBLE [100]
FSTP TBYTE [100]
```

]

The DB and DW [ENH: and DD] pseudoinstructions, which are used to assemble byte and word values. These pseudoinstructions are normally used to set aside space for and initialize data. They can also be used to synthesize instructions that DEBUG cannot assemble. Note that DEBUG will not allow a trailing comment on the same line as one of these pseudoinstructions.

The word ORG, used to set (or change) the starting address for the assembly of instructions. Note that ORG may not be supported by early versions of MS-Debug.

DEBUG checks for errors as it assembles each statement. If an error is detected, DEBUG stops assembling the statement and displays "^ Error" on the line following the statement. The position of the caret indicates the location in the statement where DEBUG detected the error.

DEBUG scripts

Because DEBUG uses the standard input and output devices, and because these devices can be redirected, DEBUG can be made to input from and/or output to a text file. This makes it possible to place DEBUG commands and statements in a text file, a so-called DEBUG script, and submit the file to DEBUG for processing. This batch file can be used to automate the process:

```
ASM.BAT:
@echo off
if "%1" == "" goto noarg
if exist %1.txt goto defext
if "%2" == "l" goto list1
if "%2" == "L" goto list1
debug < %1 | more
goto end
:list1
debug < %1 > %1.LST
goto end
:defext
if "%2" == "l" goto list2
if "%2" == "L" goto list2
debug < %1.txt | more
goto end
:list2
debug < %1.txt > %1.LST
goto end
:noarg
echo.
echo Assembles DEBUG script files
echo.
echo Usage:
echo.
echo   ASM filename[.extension] [L]
echo.
echo Extension defaults to .TXT
echo L sends output to filename.LST
:end
```

This is an example script for a Hello World program (in this and the following examples, the leading Name commands are used to place comments in the script outside of assembly mode without causing DEBUG to return an error):

```
N HELLO.TXT
N
N This is a DEBUG script that will generate HELLO.COM.
N
N HELLO.COM
```

```
A
  mov  ah,9
  mov  dx,109
  int    21
  int    20
  db     'Hello World$'

RCX
015
W
Q
```

Because DEBUG is not a symbolic assembler, you must code all operands that represent memory addresses manually. These memory addresses include data addresses and jump, loop, and call destination addresses. To do this reasonably you must assemble your code twice. In the script file for the first assembly, use a place holder value for each of these operands. Note that the placeholder values must be selected such that all displacements will be within the allowable range. Using the output from the first (error free) assembly, replace the placeholders in the script file with the actual addresses, and do the final assembly.

This is the output for the first assembly of the Hello World program, using place holder values that were essentially guesses:

```
-N HELLO.TXT
-N
-N This is a DEBUG script that will generate HELLO.COM.
-N
-N HELLO.COM
-A
1078:0100       mov     ah,9
1078:0102       mov     dx,110
1078:0105       int     21
1078:0107       int     20
1078:0109       db      'Hello World$'
1078:0115
-RCX
CX 0000
:020
-W
Writing 00020 bytes
-Q
```

This is another DEBUG script that can be used to write a boot sector to a diskette:

```
N LOADIT.TXT
N This is a DEBUG script that will copy BOOTCODE.BIN
N to the first sector of drive A. Note that this will
N work only if the OS recognizes the diskette as
N having a valid format.
N BOOTCODE.BIN
L 100
W 100 0 0 1
Q
```
