

ASSEMBLY LANGUAGE ABBREVIATED

Mark E. Donaldson

REGISTERS

GENERAL PURPOSE REGISTERS

Also called data registers, the general purpose registers are used for arithmetic and data movement. Each register can be addressed as either a 16 bit (or 32 bit) or 8 bit (or 16 bit) value. For example, the AX register is a 16 bit register. Its upper 8 bits are called AH and its lower 8 bits are called AL. Bit 0 in AL corresponds to bit 0 in AX and bit 8 in AH corresponds to bit 8 in AX. The upper halves of the 32 bit registers do not have names. To put a 16 bit value into the upper half of a 32 bit register, you must first put the value in the lower half (such as AX) and then shift the bits leftward into the upper half of the register.

AX (accumulator) – AX is the accumulator register because it is favored by the CPU for arithmetic operations.

BX (base) – BX can hold the address of a procedure or variable. Can also perform arithmetic and data movement.

CX (counter) – CX acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement CX.

DX (data) – DX has a special role in multiply and divide operations. When multiplying, for example, DX holds the high 16 bits of the product.

EAX – 32 bit AX.

EBX – 32 bit BX.

ECX – 32 bit CX.

EDX – 32 bit DX.

SEGMENT REGISTERS

The segment registers are used as base locations for program instructions, data, and the stack. All references to memory involve a segment register used as a base location.

CS (code segment) – CS holds the base location of all executable instructions (code) in the program.

DS (data segment) – DS is the default base location for variables. The CPU calculates their locations using the segment value in DS.

SS (stack segment) – SS contains the base location of the stack.

ES (extra segment) – ES is an additional base location for memory variables.

FS (x segment) – FS is an additional 32 bit base location for memory variables.

GS (x segment) – GS is an additional 32 bit base location for memory variables.

INDEX REGISTERS

Index registers contain the offsets of data and instructions.

BP (base pointer) – BP contains an assumed offset from the SS register, as does the stack pointer. BP is often used by a subroutine to locate variables that were passed on the stack by a calling program.

SP (stack pointer) – SP contains the offset of the top of the stack. SP and SS combine to form the complete address of the top of the stack.

SI (source index) – SI takes from the string movement instructions in which the source string is pointed to by the SI register.

DI (destination index) – DI acts as the destination for string movement.

EBP - 32 bit BP.

ESP - 32 bit SP.

ESI - 32 bit BSI

EDI - 32 bit DI.

STATUS AND CONTROL REGISTERS

IP (instruction pointer) – IP always contains the offset of the next instruction to be executed within the current code segment. IP and CS combine to form the complete address of the next instruction.

FLAGS – FLAGS is a special register with individual bit positions assigned to show the status of the CPU or the results of arithmetic operations.

EFLAGS – 32 bit FLAGS.

EIP - 32 bit IP.

FLAGS

CONTROL FLAGS

Individual bits can be set in the FLAGS register to control the CPU operation.

DF (direction) – DF affects block data transfer instructions such as MOVSB, CMPSB, and SCASB. The flag values are 1 = up and 0 = down.

IF (interrupt) – IF dictates whether or not system interrupts can occur. The flag values are 1 = enabled and 0 = disabled and are manipulated by the CLI and STI instructions.

TF (trap) – TF determines whether or not the CPU is halted after each instruction. When this flag is set, a debugging program can let a programmer single step (trace) through a program. The flag values are 1 = on and 0 = off. The flag can be set by the INT3 instruction.

STATUS FLAGS

The Status flags reflect the outcomes of arithmetic and logical operations.

CF (carry) – CF is set when the result of an unsigned arithmetic operation is too large to fit the destination. For example, if the sum of 200 and 58 were stored in the 8 bit register AL, the result would overflow the register and the

Carry flag would equal 1. The flag values are 1 = carry and 0 = no carry.

OF (overflow) – OF is set when the result of a signed arithmetic operation is too wide (too many bits) to fit into the destination. For example, if the sum of –128 and –2 were placed in an 8 bit BL register, the Overflow flag would be set. The Overflow flag values are 1 = overflow and 0 = no overflow.

SF (sign) – SF is set when the result of an arithmetic or logical operation generates a negative result. Because a negative number always has a 1 in the highest bit position, the Sign flag is always a copy of the destination's sign bit. The flag values are 1 = negative and 0 = positive.

ZF (zero) – ZF is set when the result of arithmetic or logical operation generates a result of zero. The flag is used primarily by jump and loop instructions to allow branching to a new location in a program based on the comparison of two values. The flag values are 1 = zero and 0 = not zero.

Auxiliary Carry – Auxiliary Carry is set when an operation causes a carry (or borrow) from bit 3 to bit 4 of an operand. The flag values are 1 = carry and 0 = no carry.

Parity – Parity reflects the number of 1 bits in the result of an operation. If there are an even number of bits, the Parity is even. If there is an odd number of bits, the Parity is odd. This flag is used by the operating system to verify memory integrity and by communications software to verify the correct transmission of data.

ASSEMBLING, LINKING, AND DEBUGGING

TURBO ASSEMBLER (TASM)

```
tasm //n/z hello.asm      ; assemble
tlink /3/m/v hello       ; link
td hello                  ; debug
```

l = listing file
n = no symbol table
z = display source lines with errors
3 = use 32 bit registers
m = linker to create map file
v = includes debug information in executable
h = display list of command line options

MICROSOFT ASSEMBLER (MASM)

```
ml /Zi hello.asm          ; assemble and link
ml /Zi /F1 /Fm hello.asm  ; assemble and link
cv hello                  ; debug
```

To link to a library file with Masm:
ml /Zi /F1 /Fm file.asm /link /co e:\masm611\lib\ Irvine

Zi = produces object file with debugging information
F1 = listing file
Fm = map file
Zm = necessary for masm511
Irvine.lib = library file name

Batch files asmTasm and asmMasm can be used to control assembly and linking process:

```
asmTasm hello
asmMasm hello
```

DATA ALLOCATION DIRECTIVES

Mnemonic	Description	Bytes	Attribute
DB	Define Byte	1	Byte
DW	Define Word	2	Word
DD	Define Doubleword	4	Doubleword
DF, DP	Define far Pointer	6	Far Pointer
DQ	Define Quadword	8	Quadword
DT	Define Tenbytes	10	Tenbyte

DB allocates storage for one or more 8 bit values. **DW** creates storage for one or more 16 bit values. **DD** allocates storage for one or more 32 bit doublewords.

```
char1      db 'A'
char2      db 'A'-10
signed1    db -128
signed2    db +127
unsigned1  db 255
```

```
val1      db ?
list      db 10, 20, 30, 40
Cstring   db "Good afternoon", 0
Pstring   db 14, "Good afternoon"
LongString db "This is a long string that "
          db "takes more than one line."
```

```
Atable    db 20 dup(0)
Rstring   db 4 dup("ABC")
```

```
unsigned1  dw 65535
list       dw 256, 258, 259
pointer1   dw list
```

```
signed_val dd, 100h
pointer2   dd subroutine1
```

SYMBOLIC CONSTANTS

Equal Sign (=) – Known as the redefinable equate, the equal sign directive creates an absolute symbol by assigning the value of a numeric expression to the name. Allocates no storage and all occurrences of name are replaced by expression. Can be redefined any number of times.

```
prod = 10 * 5
string = "XY"
maxInt = 7fffh
count = 5
```

EQU – EQU assigns a symbolic name to a string or numeric constant. A symbol defined with EQU cannot be redefined later in the program. String equates may be enclosed in angle brackets (<...>) to ensure their interpretation as string expressions.

```
maxint equ 32767
```

```
count equ 10 * 20
float1 equ <2.345>
```

TEXTEQU – TEXTEQU creates what is called a text macro. You can assign a sequence of characters to a symbolic name and then use the name later in the program.

```
continueMsg textequ <"Continue (Y/N)?">
```

```
.data
prompt1 db continueMsg
```

```
.data
myString db "A string",0
.code
p1 textequ <offset myString>
mov bx,p1      ; bx = offset myString
p1 textequ <0>
mov si,p1      ; si = 0
```

INSTRUCTIONS

DATA TRANSFER

MOV (move data) – MOV copies data from one operand to another. The sizes of both operands must be the same (a 16 bit register must be moved to a 16 bit memory location).

```
mov reg,reg      ; reg = register
mov mem,reg      ; mem= memory address
mov reg,mem
mov reg,immed    ; immed = immediate data
mov reg,immed
mov ax,[si]
mov eax,ebx
mov cl,20h
mov si,offset var1
mov dl,'X'
mov ax,(40 * 50)
```

XCHG (exchange data) – XCHG exchanges the contents of two registers, or the contents of a register and a variable.

```
xchg reg,reg     ; reg = register
xchg reg,mem     ; mem= memory address
xchg mem,reg
exch var1,bx    ; exch memory oper with register
```

MOVZX (move with zero-extend) – MOVZX, for the 80386, moves an 8 bit or 16 bit operand into a larger 16 bit or 32 bit destination register.. The unfilled bits in the destination register are cleared to Zero.

MOVSX (move with sign-extend) – MOVSX, for the 80386, moves and sign extends the source operand into the upper half of the destination register.

ARITHMETIC

INC (increment) and DEC (decrement) – INC and DEC add 1 or subtract 1 from a single operand. Destination can be a register or memory operand. All status flags are affected except the Carry flag.

```
inc al
dec bx
inc membyte
dec byte ptr membyte
dec memword
inc word ptr memword
```

ADD (add) – ADD adds a source operand to a destination operand of the same size. Source is unchanged and destination is assigned the sum. The sizes of the operands must match and no more than one operand can be a memory operand. A segment register cannot be the destination. All status flags affected.

```
add cl,al
add bx,1000h
add var1,ax
add dx,var1
add var1,10
add dword ptr memVal,ecx
```

SUB (subtract) – SUB subtracts a source operand from a destination operand. The sizes of the two operands must match and only one can be a memory operand. Inside the CPU, the source is first negated and then added to the destination.

```
sub eax,12345h
sub cl,al
sub edx,eax
sub bx,1000h
sub var1,ax
sub dx,var1
sub var1,10
```

XADD (exchange and add) – XADD, for the 80486, adds the source and destination operands and stores the sum in the destination. At the same time, the original value in the destination is moved to the source operand.

```
mov ax,1000h
mov bx,2000h
xadd ax,bx      ; AX = 3000h, BX = 1000h
```

JMP AND LOOP

JMP (jump) – JMP tells the CPU to continue execution at a different location. The location must be identified by a label, which is translated by the assembler into an address. There are three formats:

SHORT: Jump to a label in the range –128 to +127 bytes from the current location (in the same code segment). An 8 bit signed value is added to IP.

NEAR PTR: Jump to a label anywhere in the current code segment. A 16 bit displacement is moved to IP.

FAR PTR: Jump to a label in another segment. The label's segment address is moved to CS, and its offset is moved to IP.

LOOP (loop) – Loop is used to repeat a block of statements a specific number of times. CX is automatically used as a counter and is decremented each time the loop repeats.

```
mov cx,5      ; cx is the loop counter
start:
```

```
loop start    ; jump to start
```

LOOPD (loop doubleword) – LOOPD is for the 80386 processor and tells the assembler to use the 32 bit ECX register as a counter.

LOOPW – (loopword) – LOOPW is used to override the default use of the ECX register as counter when assembling in 32-bit mode.

MEMORY MODELS

Model	Description
Tiny	Code and data combined must be less than 64K. Creates a COM program.
Small	Code <= 64K, data <= 64K. One code segment, one data segment.
Medium	Data<= 64K, code any size. Multiple code segments, one data segment.
Compact	Code <= 64K, data any size. One code segment, multiple data segments.
Large	Code > 64K, data > 64K. Multiple code and data segments.
Huge	Same as the Large model, except that individual variables such as arrays may be larger than 64K.
Flat	No segments. 32-bit addresses are used for both code and data. Protected mode only.

BASIC MEMORY CONCEPTS

Collective Terms For Memory			
NAME		SIZE	
Technical	Assembler	Decimal	Hex
Byte	BYTE	1	01H
Word	WORD	2	02H
Double word	DWORD	4	04H
Quad word	QWORD	8	08H
Ten byte	TBYTE	10	0AH
Paragraph	PARA	16	10H
Page	PAGE	256	100H
Segment	SEGMENT	65,536	10000H

SEGMENT:OFFSET

In x86 CPUs, **memory addresses are composed of two parts: the segment address and the offset.** These two are added together to produce the "real" address of the memory location, by shifting the segment address one hex digit to the left (which is the same as multiplying it by 16, since memory addresses are expressed in hexadecimal notation) and then adding the segment address to it. The address itself is often referred to using the notation **segment:offset**.

The standard way to refer to **C8000h** is **C000:8000**. To get to the linear address you take **C000**, shift it one digit to the left to get **C0000**, and then add **8000** to get **C8000**. However, **C800:0000** results in the same linear address.

The Megabyte - The 8088 and 8086 CPU's can see a full megabyte of memory. They have 20 address pins and can pass a full 20 bit address (One megabyte) to the memory system. The address of a byte in a memory bank is just the number of that byte starting from zero. The addresses in a megabyte of memory run from 00000H to 0FFFFFFH. A megabyte of memory is some arrangement of memory chips within the computer, connected by an address bus of 20 lines.

The Paragraph - A paragraph is a measure of memory equal to 16 bytes. The term paragraph is almost never used except in connection with the places where segments begin. **Any memory address evenly divisible by 16 is called a paragraph boundary.** The first paragraph boundary is address 0. The second is address 10H. The third address 20H, and so on (10H is equal to decimal 16). Any paragraph boundary may be considered the start of a segment. An assembly language program may make use of only four or five segments, but each of those **segments may begin at any of the 65,536 paragraph boundaries existing in the 8088/8086 megabyte of memory.** The are 64K different paragraph boundaries where a segment may begin.

The Segment - Although the CPU can see a full megabyte of memory, it is constrained to look at that megabyte through 16 bit blinders. In other words, it can only look at a consecutive 65,536 bytes at one time (for 16 bit registers). A **segment is a region of memory that begins on a paragraph boundary and extends for some number of bytes less than or equal to 64K (65,536).** A segment may be up to 64K bytes in size but it doesn't have to be. You define a segment by starting where it begins.

Each paragraph boundary has a number. Because a segment may begin at any paragraph boundary, the number of the paragraph boundary at which a segment begins is called the **segment address** of that particular segment. When you see the term segment address, keep in mind that each segment address is 16 bytes (one paragraph) farther along in memory than the segment address before it. In short, **segments may begin at any segment address.** **There are 65,536 segment addresses evenly distributed across the full megabyte of memory, 16 bytes apart.**

OPERATORS

Operator	Description
.TYPE	Returns a byte that defines the mode and scope of an expression. The result is bit mapped and is used to show whether a label or variable is program related, data related, undefined, or external in scope.
+, -, *, /	Addition, subtraction, multiplication, and division.
AND, OR, NOT	Bitwise operations on constant integers.
EQ, NE, LT, LE, GT, GE	Relational operators. Assembler returns a value of 0FFFFh when a relation is true or 0 when it is false.
HIGH	Returns the high 8 bits of a constant expression.
HIGHWORD	Returns the high 16 bits of a 32 bit operand (MASM only).

LENGTH	Returns the number of byte, word, dword, qword, or terabyte elements in a variable. This is meaningful only if the variable is initialized with the DUP operator.
LOW	Returns the low 8 bits of a constant expression.
LOWWORD	Returns the low 16 bits of a 32 bit operand (MASM only).
MASK	Returns a bit mask for the bit positions in a field within a variable. A bit mask preserves just the important bits, setting all others equal to zero. The variable must be defined with the RECORD directive.
MOD	Modulus operator. Returns the integer remainder of a division operation.
OFFSET	Returns the offset of a label or variable from the beginning of its segment.
PTR	Specifies the size of an operand, particularly when its size is not clear from the context.
SEG	Returns the segment value of an expression, whether it be a variable, a segment/group name, a label, or any other symbol.
SHORT	Sets a label's attribute to SHORT. Often used in JMP instructions.
SIZE	Returns the total number of bytes allocated for a variable. This is calculated as the LENGTH multiplied by the TYPE. The type of a near label is FFFFh, and the type of a far label is FFFEh.
Field (.)	The name following (.) identifies a field within a predefined structure by adding the offset of the field to the offset of the variable. The format is variable.field.
THIS	Creates an operand of a specified type as the current program location. The type can be any of those used with the PTR operator or the LABEL directive.
TYPE	Returns an integer that represents either the size of a variable or its type. For example, the TYPE of a word variable is 2.
WIDTH	Returns the number of bits of a given field within a variable that has been declared with the RECORD directive.

DIRECTIVES

Directive	Description
RECORD	
LABEL	Allows you to insert a label and give it a size attribute, without allocating any storage. Any of the standard size attributes can be used. Often used as an alias. A way to get around the assembler's requirement that the size attribute of a variable must match the other operand in an instruction. A typedef.
EVEN	Aligns the next instruction in the code segment to an even 16-bit offset.
EVENDATA	Inserts a null byte (0) before an array.
ENUM	Tasm only. Allows you to define a set of enumerated constants and automatically assign an integer to value to each one. By default, the first constant is assigned a value

	of 0, the second a value of 1, the third a value of 3, etc.
.MODEL	Indicates memory model type and size.
.STACK	Sets aside stated number bytes of stack space for program.
.DATA	Marks the beginning of data segment where variables are stored.
.CODE	Marks the beginning of code segment where executable instructions are located.
.8086	Enables assembly of 8086, 8087, and 8088 instructions. Disables assembly of instructions for the 80186 and latter processors.
.186	Enables assembly of 80186 instructions and disables assembly of instructions all latter processors.
.286	Enables assembly of 80286 instructions and disables assembly of instructions all latter processors.
.386	Enables assembly of 80386 instructions and disables assembly of instructions all latter processors.
.486	Enables assembly of 80486 instructions and disables assembly of instructions all latter processors.
.586	Enables assembly of nonprivileged Pentium instructions.
.287	Enables assembly of floating point instructions for the 80287 math coprocessor.
.387	Enables assembly of floating point instructions for the 80387 math coprocessor.
.286P	Enables assembly of privileged mode instructions of the 80286 and disables assembly of privileged mode instructions all latter processors.
.386P	Enables assembly of privileged mode instructions of the 80386 and disables assembly of privileged mode instructions all latter processors.
.486P	Enables assembly of privileged mode instructions of the 80486 and disables assembly of privileged mode instructions all latter processors.
.586P	Enables assembly of privileged mode instructions of the 80586 and disables assembly of privileged mode instructions all latter processors.
EXTRN	Used to identify source names that exist outside the current source file.
PROC	Begin procedure.
ENDP	End of procedure.
END	End of program assembly.
PAGE	Set a page format for the listing file.
TITLE	Title of the listing file.

Data Types For EXTRN Directive	
Type	Description
ABS	A constant defined with EQU or =
PROC	Default type for procedure
NEAR	Name is in the same segment
FAR	Name is in a different segment

BYTE	Size is 8 bits
WORD	Size is 16 bits
DWORD	Size is 32 bits
FWORD	Size is 48 bits
QWORD	Size is 64 bits
TBYTE	Size is 10 bytes

DEBUG COMMANDS

? (Help) – Press ? at the debug prompt to see a list of all commands.

A (Assemble) – Assemble a program into machine language. If only the offset portion of address is supplied, it is assumed to be an offset from CS. Command format:

A
A *address*

Examples:

A Assemble from current location
A 100 Assemble at CS:100h
A DS:2000 Assemble at DS:2000h

C (Compare) – Compares bytes between the specified range with the same number of bytes at a target address. Command format:

C *range address*

D (Dump) – Displays memory on the screen as single bytes in both hexadecimal and ASCII. If no address or range is given, the location begins where the last D command left off, or at location DS:0 if the command is being typed for the first time. If address is specified, it consists of either a segment-offset address or just a 16-bit offset. Range consists of the beginning and ending addresses to dump. The default segment is DS, so the segment value may be left out unless you want to dump an offset from another segment location. A range may be given, telling Debug to dump all bytes within the range. Command formats:

D
D *address*
D *range*

Examples:

D Dump 128 bytes from last reference
D 150 15A Dump DS:0150 through 015A
D SS:0 5 Dump bytes at offsets 0-5 from SS
D 915:0 Dump 128b offset zero from seg 0915h
D 0 200 Dump offsets 0-200 from DS
D 100 L 20 Dump 20b starting at offset 100h from DS

E (Enter) – Places individual bytes in memory. You must supply a starting memory location where the values will be stored. If only an offset is entered, the offset is assumed to be from DS. Otherwise, a 32-bit address may be entered or another segment register may be used. Command formats:

E *address* Enter new byte value at address
E *address list*

Example – To begin entering hexadecimal or character data at DS:100 type E 100. Press space bar to advance to the

next byte, and press the Enter key to stop. To enter a string into memory starting at location CS:100 type E CS:100 "This is a string."

F (Fill) – Fills range of memory with a single value or list of values. The range must be specified as two offset addresses or segment-offset addresses. Command format:

F *range list*

Examples:

F 100 500, ' ' Fill locations 100 through 500 with spaces
F 100 L 20 'A' Fill 20h bytes with letter A, starting at 100

G (Go) – Execute the program in memory. You can specify a breakpoint causing the program to stop at a given address. Breakpoint is a 16-bit or 32-bit address at which the processor should stop, and startAddr is an optional starting address for the processor. If no breakpoints are specified, the program runs until it stops by itself and returns to Debug. Up to 10 breakpoints may be specified. Command formats:

G
G *breakpoint*
G = *startAddr breakpoint*
G = *startAddr breakpoint1 breakpoint2*

H (Hexarithmic) – Performs addition and subtraction on two hexadecimal numbers. Command format:

H *value1 value2*

L (Load) – Loads a file (or logical disk sectors) into memory at a given address. To read a file, you must first initialize its name with the N (Name) command. If address is omitted, the file is loaded at CS:100. Debug sets BX and CX to the number of bytes read. Command format:

L
L *address*
L *address drive firstsector number*

Examples:

L Load named file into memory at CS:0100
L DS:0200 Load named file into memory at DS:0200
L 100 2 A 5 5 sectors drive C starting at sector 0Ah
L 100 0 0 2 2 sectors at CS:100 from A at sector 0

M (Move) – Copies a block of data from one memory location to another. Range consists of the starting and ending locations for the bytes to be copied. Address is the target location to which the data will be copied. All offsets are assumed to be from DS unless specified otherwise. Command format:

M *range address*

N (Name) – Initializes a filename (and file control block) in memory before using the L (Load) or W (Write) commands. Command format:

N [*d:*][*filename*][*.ext*] N *b:myfile.dta*

P (Proceed) – Executes on or more instructions or subroutines. Whereas the T (Trace) command traces into

subroutine calls, the P command simply executes subroutines. Also, LOOP instructions and string primitives instructions (SCAS, LODS) are executed completely up to the instruction that follows them. Command formats:

P
P =address
P =address number

Examples:
P =200 Execute a single instruction at CS:0200
P =1506 Executes 6 instr starting at CS:0150
P6 Execute next 5 instructions

Q (Quit) – Quits Debug and returns to DOS.

R – (Register) – Used to display the contents of one register allowing it to be changes, to display registers, flags, and the next instruction to be executed, and to display the eight flag settings allowing any or all of them to be changed. Commands formats:

R
R register

Examples:
R Display contents of all registers
R IP Display contents of IP & prompt new value
R CX Display contents of CX register
R F Display all flags & prompt new flag value

S (Search) – Searches a range of addresses for a sequence of one or more bytes. Command format:

S range list

Examples:

S 100 1000 0D Search DS:100-DS:1000 for 0Dh
S 100 1000 CD, 20 Search for sequence CD 20
S 100 9FFF "COPY" Search for word "COPY"

T (Trace) – Executes one or more instructions starting at either the current CS:IP location or at an optional address. The contents of the registers are shown after each instruction is executed. Command formats:

T
T count
T =address count

Examples:
T Trace the next instruction
T 5 Trace the next five instructions
T =105 10 Trace 16 instructions starting at CS:105

U (Unassemble) – Translates memory into assembly language mnemonics. This is also called disassembling memory. If you don't supply an address, Debug disassembles from the location where the last U command left off. If the command is used for the first time after loading Debug, memory is unassembled from location CS:100. Command formats:

U

U startAddr
U startAddr endAddr

Example:
U Disassemble the next 32 bytes
U 0 Disassemble 32 bytes at CS:0
U 100 108 Disassemble bytes CS:100 to CS:108

W (Write) – Writes a block of memory to a file or to individual disk sectors. To write to a file, its name must first be initialized with the N (Name) command. The command format is identical to the L (Load) command format:

W address
W address drive firstsector number

Examples:

W Write 20h bytes to file starting at CS:100
W 0 Write from location CS:0 to the file
W Write named file from location CS:0100
W DS:0200 Write name file from location DS:0200