Cromemco

MACRO ASSEMBLER

INSTRUCTION MANUAL ADDENDUM

CROMEMCO, INC.
280 Bernardo Avenue
Mountain View, CA 94043

Part No. 023-4001 January, 1980

This manual was produced on a Cromemco System Three Computer using the SCREEN Editor. The edited text was formatted using the Cromemco Word Processing System Formatter. Final camera-ready copy was printed on a Cromemco 3355A printer.

TABLE OF CONTENTS

INTRODUCTION

The Cromemco Macro Assembler version 3.05 and higher has several significant enhancements. These enhancements are covered in the first part of this addendum.

Page numbers refer to the Cromemco Macro Assembler Manual (part number 023-0039) as published in October 1978.

The second part of this addendum is intended to replace Part III (pp. 117-138) of the same manual. This section of the addendum covers the use of the Debug program (version 00.14 and higher) which has also been significantly enhanced.

OPTIONS SPECIFIED WHEN CALLING ASMB (pp. 19-27)

LISTON, LISTOFF

These options, when given in the command line which calls the Assembler, override the list ON and OFF pseudo-ops (pp.52-53). The LISTOFF option is useful if the programmer desires a cross reference listing without a program listing.

TEXT, NOTEXT

The NOTEXT option causes the Assembler to suppress the printing of the generated object code associated with DB statements.

TRUNC = nn

This option is similar to the WIDTH option (pp. 25-26) except that lines longer than nn characters are truncated and not wrapped around. Lines longer than 128 characters are truncated by default unless the WIDTH option is specified.

NOXREF

This option suppresses the generation and printing of the cross reference listing. This may be useful if the default has been changed to XREF on.

DATE=mmddyy
TIME=hhmmss

These options cause the specified date and time to be included in the program listing which is generated by the Assembler. Note that month, day, year, hour, minute, and second must each be specified as two digit numbers as shown above.

If time and date are set in CDOS the Assembler will use these values.

NAMES (LABELS) (p. 30)

Symbol names may now contain up to eight significant characters except for ENTRYs and EXTRNs which may contain up to seven significant characters. The following special characters may now be used anywhere in a symbol name:

. (period)
$ (dollar sign)
_ (underscore)
? (question mark)
@ (at sign)

CONDITION CODES

In order to allow the programmer to make listings more readable several new Condition Codes have been defined. These codes may be used in place of those specified in the Zilog and Mostek Assembly Language Programming Manuals.

CODE	CONDITION	RELEVANT FLAG	EQUIVALENT CONDITION
valid for JP, RET, and CALL instructions:			
V	overflow	P/V	PE parity even
NV	no overflow	P/V	PO parity odd
valid for JR, JP, RET, and CALL instructions:			
EQ	equal	Z	Z zero
NE	not equal	Z	NZ non zero
LT	less than	C	C carry
GE	greater or equal	C	NC no carry

MODIFYING THE DEFAULT VALUE OF OPTIONS

The Default value of various Assembler options can be changed by modifying locations 103H to 109H. These changes may be accomplished by using the Cromemco DEBUG program and saving the modified file.

Hex Location	Current bit	Default	Option
103		3AH	PAGE=
104		80H	{WIDTH= {TRUNC=
105		0	TOP=
106	2	1	XREF
	3	0	SYMB
	4	1	RANGE
	5	0	PARITY
107	1	0	NOGEN
	2	0	NOCOND
	3	0	OPCODE
	5	0	COND
	6	0	GEN
108	2	0	TEXT
	3	1	NOTEXT
	4	0	LISTON
	5	0	LISTOFF
	6	1	TRUNC
109		7	(number of chars for ENTRY & EXTRN) should only be 6 or 7

EXPRESSIONS AND OPERATORS (pp. 34-36)

NEW OPERATORS

The following new operators have been added to those described on page 35 of the Cromemco Macro Assembler Manual:

<> not equal, same as NE

>= greater than or equal, same as GE

<= less than or equal, same as LE

% modulus, same as MOD

<< shift left logical, same as SHL

>> shift right logical, same as SHR

& logical and, same as AND

| logical or, same as OR

~ logical not, same as NOT

^ set bit as specified by the expression following the operator. This is a unary operator and may alter bit 0-15 to form an integer constant. This operator has the highest precedence.

Low return the low byte of the following expression. This is a unary operator with the lowest precedence.

High return the high byte of the following expression. This is a unary operator with the lowest precedence.

STRING COMPARISON

This new feature finds the most use in Macro definitions. An ASCII string comparison is of the following format:

"string-1" relop "string-2"

where relop can be any one of:

```
=
<>
>=
<=
<
>
```

and string-1 and string-2 may be of any length. If one string is shorter than the other, the shorter one is left justified and padded with nulls.

EXTERNAL ± OFFSET

A label which is declared external (pp. 47-49) may now be used with an absolute expression (or a constant) added to or subtracted from it. Thus, the following would be a legal section of code:

```
            extrn   table
disp:       equ     3
             .
             .
            ld      a,(table+disp)
             .
             .
```

UNDEFINED EXPRESSIONS

Undefined Expressions for IF, EQU, DEFL, ORG, REPT, STRUCT, and DEFS now give errors on pass one of an assembly. This avoids the generation of a complex phase error during pass two.

NEW PSEUDO-OPS (pp. 39-63)

SUBTITLE

In addition to a TITLE pseudo-op (pp. 56-57) the programmer may now specify a subtitle. The subtitle is specified in the same manner as the TITLE using one of the following equivalent pseudo-ops:

```
TITLE2
SUBTTL
```

LIST - TEXT, NOTEXT

In the same manner as the LIST - ON, OFF pseudo-ops (pp. 52-53), the LIST - TEXT, NOTEXT pseudo-ops cause the listing of generated object code associated with DB statements to be suppressed.

*MACLIB

The Macro Library pseudo-op defines a Macro Library in the same manner as the MACRO= option (pp. 22-23). Up to 16 different libraries may be defined during one assembly. The format of this pseudo-op is:

```
*MACLIB file-ref
```

where file-ref is a file reference composed of a file name and optionally a disk drive specifier and file name extension.

*RELLIB

The relocatable library pseudo-op defines a library of relocatable routines which will be searched at link time for unresolved externals. The format of this pseudo-op is:

```
*RELLIB file-name
```

where file-name is a file name containing no more than 7 characters. The file must be on the disk in the current drive. No file name extension may be

specified in the *RELLIB pseudo-op but the file must have a file name extension of REL on the disk.

STRUCT

The STRUCTure pseudo-op defines a series of labels as offsets from a base. The format is as follows:

```
            STRUCT  <exp>
NAME1:      DS      <size>
NAME2:      DS      <size>
             .
             .
             .
            MEND
```

where exp is an expression denoting the initial offset (usually zero) and size is the size of each element of the structure. Note that the DS pseudo-ops within the STRUCT-MEND definition do not reserve any space.

For example, the following structure:

```
            struct  20
x:          ds      4
y:          ds      2
z:          ds      1
size:       ds      0
            mend
```

is the functional equivalent of:

```
x:          equ     20
y:          equ     24
z:          equ     26
size:       equ     27
```

Each of these sections of code define offsets of the labels x, y, z, and size. Notice that in the example, size is automatically set equal to the maximum offset value.

Remember that the DS pseudo-op does not reserve storage area only within the bounds of a STRUCT-MEND definition. In all other cases, DS functions as described in the Assembler Manual (pp. 42-43).

Given either of the above examples, storage space for three different structures could be reserved by

the following code:

```
stra:     ds      size
strb:     ds      size
strc:     ds      size
```

Then the following code will load the A register with the contents of the second member of STRC:

```
ld        ix,strc
ld        a,(ix+y)
```

Using the STRUCT form has the advantage of automatically computing the offset and maximum offset while allowing the programmer to add or delete elements of the structure without re-computing the offset for each original element.

DEFV, DV

These pseudo-ops perform the same function as DEFL and DL (pp. 40-41) except that a label defined by DEFV or DV will not appear in the cross reference listing. A label defined by DEFV or DV may be redefined at a later point in the program.

CONMSG

This pseudo-op will display a message on the console during the second pass of the assembler.

The format is:

```
CONMSG <any message>
```

GLOBAL

The GLOBAL pseudo-op acts as an EXTRN (pp. 47-49) if the label following it is undefined and as an ENTRY (pp. 45-46) if it is defined in an assembly module. This finds a great many applications when used in conjunction with the *INCLUDE pseudo-op.

ELSE

The ELSE pseudo-op may be included between the IF and ENDIF statements (pp. 74-76) to form an if-then-else structure for conditional assembly.

MACROS (pp. 65-74)

COMMENTS

Any comments in a Macro which are preceded by two semicolons (as opposed to the usual one) will be thrown away by the Assembler. This will save memory during the assembly and will also create a shorter PRN file.

EXITM

When this instruction is encountered during the expansion of a Macro the balance of the Macro will not be expanded. This can be used to advantage in conjunction with conditional assemblies.

ARGUMENT SUBSTRINGS

Any character or group of characters in a Macro argument (parameter) may be accessed in a Macro definition. Assuming that a Macro has an argument defined as #ARG any character or characters may be referenced by one of the two forms:

```
#ARG(first-pos,last-pos)
#ARG(first-pos)
```

where first-pos and last-pos are the desired character indexes within the argument string. Position expressions are:

```
 1 = first character
 2 = second character
 .
 .
 .
-2 = second to last character
-1 = last character
```

This addressing scheme allows the same character to be addressed either by its position relative to the beginning or end of the string.

OMACRO

This represents an opcode Macro. A Macro defined by OMACRO will be placed in the OPCODE XREF table instead of the SYMBOL XREF table.

REPEAT EXPANSIONS

The Repeat Expansions allow the programmer to write repetitive code in a more structured fashion so that it may be more easily written, understood, debugged, and modified. The expansions do not generate code with loops in it. They expand the code as is demonstrated in the examples.

REPT

This form will repeat the generation of a given section of code a specified number of times. The format is:

```
REPT <exp>
  .
  .
(code)
  .
  .
MEND
```

where exp is an expression whose value determines the number of times the code is to be repeated.

Example:

```
rept      256
db        0FFH
mend
```

will generate the following code:

```
db        0FFH
 .
 .
(256 times)
 .
 .
db        0FFH
```

IRP

The iterative repeat will repeat a given section of code, substituting a new value for a given argument, until it runs out of values. The format

is:

```
IRP #ARG, ARG1, ARG2, ARG3,...
  .
  .
(code)
  .
  .
MEND
```

where #ARG is the argument which takes on the value of ARG1 the first time through the loop, the value of ARG2 the second time through, etc. When the loop has been expanded one time with each of the arguments the expansion will terminate.

Example:

```
ld        hl,0
irp       #var,x,y,z
ld        (#var),hl
mend
```

will generate the following code:

```
ld        hl, 0
ld        (x),hl
ld        (y),hl
ld        (z),hl
```

IRPC

The iterative repeat with characters will repeat a given section of code, substituting a new character for a given argument, until it runs out of characters. The format is:

```
IRPC #ARG, 'character string'
  .
  .
(code)
  .
  .
MEND
```

where #ARG is the argument which takes on the value of each successive character in the character string each time through the loop. When the #ARG has taken on the value of the last character in the string the expansion will terminate.

Example:

```
      irpc    #char,'COM'
      ld      (hl), '#char'
      inc     hl
      mend
```

will generate the following code:

```
      ld      (hl),'C'
      inc     hl
      ld      (hl),'O'
      inc     hl
      ld      (hl),'M'
      inc     hl
```

LISTINGS

CROSS REFERENCE

The cross reference listing now takes the following format:

NAME	FLAGS	VALUE	DEFN	REFERENCES
	E - entry X - extrn M - multdef U - undef			
		MACRO	1234	5678

SYMBOL TABLE

Macros are now listed in the Symbol Table while Omacros (opcode Macros) are listed in the Opcode Xref Table.

MODIFYING CDOS I/O DEVICE DRIVERS

The Assembly Language source file for the CDOS I/O Drivers is now supplied on the disk with the Cromemco Macro Assembler. For most applications the programmer need not be concerned with this file. The I/O Drivers are already incorporated in CDOSGEN and this file is included only for those programmers who need to modify the Drivers.

A programmer attempting to modify the Drivers must be familiar with Z-80 Assembly Language programming, conditional assembly, the Cromemco Z-80 Macro Assembler, and the design of I/O drivers.

The file containing the CDOS I/O Drivers is called DRIVERS.Z80. This file contains switches for conditional assembly and EQUs for port assignments followed by the routines for the various devices.

The following points should be observed when modifying the Drivers:

1. The programmer must follow the instructions and notes in the source listing.

2. No tables may be moved or changed. This applies to those tables which CDOS needs and expects in certain locations.

3. All routines are preceded by a header which specifies entry and/or exit parameters, register contents, etc. These specifications must be observed as CDOS is dependent upon them.

4. If the programmer uses any of the prime registers or the IX or IY registers their value must be preserved (typically on the stack). The non-prime registers need only be preserved to the extent which they are used (as in number 3 above).

5. The CDOS stack should not be used to a depth greater than ten (approximately).

The following procedure will create a CDOS with the modified I/O Drivers as specified in the file MYDRIVER.Z80. Notice that although the procedure

must be followed step by step the names of the files may be changed as desired. The commands in the left column are given in response to the CDOS prompt while the right hand column explains the purpose of each.

SCREEN MYDRIVER.Z80	Using SCREEN or EDIT change the file DRIVERS.Z80 as is needed and then exit from the editor. Always maintain a copy of the original source file for reference. Here a copy of the file is called MYDRIVER.Z80. The Z80 file name extension is required.
ASMB MYDRIVER.@@Z HEX=0	Assemble the Drivers in HEX format with an ORG of 0. Note that the file name extension of @@Z will instruct the Assembler that the source file is located on the current disk, the object file is to be placed on the current disk, and that no print file is to be produced.
REN MYD0.HEX=MYDRIVER.HEX	Rename assembled HEX file.
ASMB MYDRIVER.@@Z HEX=100	Assemble the Drivers in HEX format with an ORG of 100H.
REN MYD100.HEX=MYDRIVER.HEX	Rename the assembled Hex file. At this point the original source file (MYDRIVER.Z80) is still present and unchanged on the

	disk.
CDOSGEN MYD0.HEX MYD100.HEX	This is the final step which generates a version of CDOS using the modified Drivers. The two HEX files are used to relocate the drivers to their final location in CDOS. All questions in CDOSGEN must be answered as usual. When CDOSGEN has finished writing the CDOS file to the disk CDOS must be booted up again.

Once the the new CDOS has been booted up the new I/O Drivers will be operational.

INTRODUCTION TO DEBUG

DEBUG ENHANCEMENTS

1. More powerful expressions.

2. Permanent break points.

3. Conditional break points.

4. New commands:
   B -break points
   C -tracing
   Q -query memory
   W -write disk file
   Z -zap memory

5. Bug fixes and changes:
   Lower case is accepted as input to the SM command.
   Area where Debug was loaded is now zeroed.
   Proper names are used for the registers.
   Initial value of SP is now top of memory.
   Relative addresses are always displayed if the @ register is not equal to 0.

LOADING DEBUG

The Cromemco Debug program makes it possible to test, debug, and trace through user programs. Debug is loaded into memory at 100H and moved to the highest memory available below CDOS.

Loading Debug is accomplished by typing one of the following commands from CDOS:

```
DEBUG
DEBUG file-ref
```

where file-ref is a file reference to the program to be tested. The CDOS jump instruction located at location 5H is changed to jump to the start of Debug. This allows locations 6H and 7H to still point to the lowest available memory location (below Debug).

The second form of the command is used to load the

file to be tested into memory. If the file name extension is HEX, then the file is read as an Intel format HEX file. A file with any other file name extension is read as an absolute binary file and loaded at location 100H. Note that Debug does not link relocatable files. If a relocatable file (with a file name extension of REL) is specified it will be loaded as if it were a binary file and will not be executable.

CONTROL CHARACTERS

Control characters are used in Debug to help enter commands. These control characters are the same ones CDOS uses.

Control-C (^C)	return to CDOS
Control-H (^H)	delete character and backspace on CRT
Control-U (^U)	delete line
Control-V (^V)	delete line
Control-X (^X)	delete character and echo
underscore(_)	delete character and backspace on CRT
RUBout (DEL)	delete character and backspace on CRT

While displaying new information (such as that which is displayed by the DM command) the following characters may be used:

Control-S (^S)	stop/start printing or listing to the console. If printing, this character will stop the printing. If already stopped by the use of a CTRL-S, this or any other character will resume the printing.
space	(or any other character) will abort the printing or console display, prompt, and wait for the next command.

COMMAND FORMAT

Debug is controlled by one and two character commands from the terminal. The format is free-form with respect to spaces. Commas may be used in place of spaces. Remember that all commands must be terminated with a carriage return. The following examples all display memory starting at location 1000H and ending at location 10FFH.

```
D1000 10FF
DM1000S100
DM 1000,10FF
D 1000,S100
```

CURRENT PROGRAM COUNTER LOCATION ($)

The current address (the current value of the program counter) may be represented by a dollar sign ($). The following command will begin program execution at the current value of the program counter and will stop execution (by the use of a temporary break point) when the program counter reaches its current value plus 3:

```
G/$3
```

COMMAND STARTING ADDRESS

The Assemble into Memory (A), Display Memory (DM or D), List in Assembler Mnemonics (L), and Substitute Memory (SM or S) commands each maintain a starting address to use if none is given with the command. This address is changed each time one of the specified commands is executed so that the next execution of the command will commence where the last one ended. When Debug is entered each of the starting addresses is set to 100H.

@ REGISTER

Debug was designed to aid the programmer in testing relocatable programs. The @ register is used to tell Debug where the module you wish to debug is located. This address can be found from the map generated by the Cromemco linking loader LINK. To change the @ register, type:

```
@
```

on the console. Debug will then display:

```
@-xxxx
```

where xxxx is the current value of the @ register. The computer will then wait for a new address. If only a carriage return is typed, the register will remain unchanged. If an address and a carriage return is typed, then the register will contain the new address. The @ register may now be used as part of an address:

```
G/@ @A3 1000
```

This is an example of the GO command. Break points will be set at the beginning of the current module, relative location A3H in the current module, and at location 1000H. This feature allows you to test a module without having to calculate absolute addresses.

The relative address is displayed in addition to the absolute address whenever addresses are displayed unless the @ register equals zero.

ADDRESS EXPRESSIONS

For additional ease in specifying addresses an expression can be used. Within these expressions, addition, subtraction, the relative address (@) register, and the current program counter location ($) may be used. If many modules are being tested, addition can be used to specify relative addresses.

EXPRESSIONS

Special Symbols:

^reg	Register value (see Summary of Register Names)
@	@ register
$	Program counter
(expr)	Contents of memory addressed by "expr"
[expr]	Used to alter order of evaluation
'xy'	Ascii value of "xy"
ddddd.	Decimal number
hhhh	Hexadecimal number

Unary Operators:

+	Positive number
-	Negative number
~	Logical Not (complement)

Binary Operators:

+	Addition
-	Subtraction
*	Multiplication
%	Division
&	Logical And
|	Logical Or

Relational Operators:

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

An expression can be used in Debug anywhere that a number is needed, i.e., for addresses, data, etc. Expressions must not contain any imbedded spaces. The precedence of operators is as follows:

1. Special symbols (highest precedence)
2. Unary operators
3. Binary operators
4. Relational operators (lowest precedence)

The order of evaluation may be altered by the use of brackets ([ ]).

Among operators of the same precedence, evaluation proceeds from left to right. For example:

10.+20.*3. equals 90. (decimal)

^DE=^HL equals 0000 if the DE register does not equal the HL register
equals FFFF otherwise

G/2321+A3 would set a break point at relative location A3H if the module is located at 2321H

SWATH OPERATOR

There are two ways to specify the address range of many commands. The first is to simply list the beginning and end addresses (and where appropriate, the destination address). For example, the first command below programs the range 0 through 13FFH into PROMs starting at location E400H. The second command displays the contents of memory between addresses E400H and E402H.

```
P0 13FF E400
DME400 E402
```

Another way to do the same thing is to use the Swath operator to specify the width of the address range, rather than explicitly stating the end address.

```
P0 S1400 E400
DM E400S3
```

ERRORS

Any errors made during command entry may be corrected by typing Control-V (^V) to abort the line or by backspacing and correcting the line. If a carriage return has already been entered and Debug detects an error, the line will not be accepted and a question mark will be displayed. The command must then be re-entered correctly.

A = ASSEMBLE INTO MEMORY

This command allows the user to enter assembly language mnemonics from the console and have them assembled into memory. The command takes the following format:

```
A
A beginning-addr
```

The first format assembles into memory starting at the default current address (initially 100H). The second format assembles starting at beginning-addr.

The user is prompted with the absolute address, relative address, and the current instruction located at that address.

Debug reads from the console and assembles the instruction into memory. The mnemonics for the various Z-80 instructions can be found in the Z-80 CPU Technical Manual published by Mostek and Zilog (Cromemco part number 023-0045). If there is no error in the instruction it is assembled into memory and the user is prompted for the next instruction. The rules for address expressions apply to the addresses in the assembler mnemonics. In the following example the @ register contains 1234H.

```
A@40
1274 0040'      NOP     ADD B
1275 0041'      NOP     CALL @93
1278 0044'      NOP     JP 1032+95
127B 0047'      NOP     .
```

If only a carriage return is typed by the user, Debug does not alter the current instruction and goes to the next instruction in memory.

The A command terminates when a line starting with a period (.) is entered. If there is an error in the input line, it will not be accepted, a question mark will be displayed, and the user will be prompted again.

B - SET or DISPLAY PERMANENT BREAK POINTS

The B command is used to set permanent break points. Using this feature, it is not necessary to repeatedly set break points when using the G command. A permanent break point will remain in effect until it is explicitly removed by the BX command.

It is possible to set a total of 12 break points including the permanent breakpoints (B command), temporary breakpoints (G command), and tracing breakpoints (T and C commands).

To display all of the currently active permanent break points enter the command:

```
B
```

To set a permanent break point enter the command:

```
B breakpoint-1 breakpoint-2...breakpoint-n
```

A break point is specified by up to four fields:

Field	Use
R	Report registers flag (optional)
addr	Memory address for break point
{cond}	Condition for break point (optional)
:count	Repeat count for break point (optional)

The report registers flag is used to display the registers each time the break point is encountered during execution. This is used in conjuction with the repeat count field.

The condition field is used to specify a conditional break point. If a condition is specified the break point will not stop the program unless the condition is true (non-zero). The condition can be any expression involving registers or memory. Each time the user program reaches the break point address the expression is evaluated. If the result is false (zero) the user program continues as if the break point did not exist.

The repeat count is used to stop the user program the nth time it reaches the break point address. The repeat count is decremented each time the

program reches the break point. If the count goes to zero, the user program will be stopped.

Note that if a condition and a repeat count are both used, the repeat count is only decremented if the condition is true.

The repeat count is never reset to its original value. The programmer must respecify the break point (using the B command) if the repeat count needs to be reset.

128 bytes are dedicated to conditional expressions so that all conditions together can contain a maximum of 128 characters.

Examples:

```
B 106
```

This will establish a permanent break point at memory location 106H. Whenever the program counter (PC) is equal to 106H program execution will stop and the registers will be displayed in standard format.

```
B 106:5
```

This variation on the first example will cause program execution to stop on the fifth execution of the instruction at 106H. The registers will be displayed only when program execution stops.

```
B R106:5
```

This time the report registers flag has been set so that each time the break point is passed the registers will be displayed. Again execution will be terminated the fifth time the instruction at location 106 is to be executed.

```
B 106 {^A=0}
```

Now a conditional permanent break point has been set. Execution will terminate and the registers will be displayed only when the program counter equals 106H and the A register equals 0.

```
B 5{[^C=9]|[^C=A]}:25
```

The final example in this section will stop program

execution when, for the 25th time, the C register equals 9H or AH and the program counter equals five. Register names are preceded by the up-arrow (caret) character while hexadecimal numbers are not; the two should never be confused. When debugging a program which is running under CDOS, this break point will have the effect of stopping program execution on the 25th system call to print or input a buffered line.

BX - DELETE PERMANENT BREAK POINTS

This command is used to delete permanent break points. It has two formats:

```
BX
BX addr-1 addr-2 ...
```

The first format will delete all permanent break points. The second format will delete the break points at each of the specified addresses.

C - TRACE OVER CALLS

```
C
C number of instructions
C {expr}
```

The Trace Over Calls command functions in a manner similar to the simple Trace command. The difference appears when a CALL or RET instruction is encountered in a multiple instruction trace.

When a CALL instruction is encountered during the execution of the C command no tracing is performed in the called subroutine. Tracing resumes when control is returned to the current subroutine at the location which is three bytes beyond the CALL instruction. The number-of-instructions counter is not decremented while program control remains in the called subroutine.

When a RET instruction is encountered during a multiple instruction trace, tracing will stop and the RET instruction will be displayed. This allows the programmer to stop execution of a subroutine before control is returned to the calling routine.

At this point the registers can be examined (and modified) before the return instruction is executed.

If a conditional RET instruction (e.g., RET Z) is encountered program execution will stop only if the RET condition is true (i.e., if the RET instruction is to be executed). Otherwise the conditional RET instruction will be treated as any other instruction and program execution will continue.

When the third form of the command is used execution will continue until the expression (expr) is true (not equal to 0). The expression will only be evaluated while program control remains within the current subroutine. The CALL and RET instructions function as described above.

CN - TRACE OVER CALLS WITH NO DISPLAY

```
CN
CN number of instructions
CN {expr}
```

The CN command is similar to the C command except that no information is displayed as the trace progresses. When the trace terminates, the standard register information is displayed.

CJ - TRACE OVER CALLS WITH JUMPS

```
CJ
CJ number of instructions
CJ {expr}
```

The Trace Over Calls with Jumps command is similar to Trace Over Calls with one addition. Break points are established before all instructions which alter the program counter (i.e., JP and JR).

The CALL and RET instructions function as they do in the Trace Over Calls command.

With this command only JP, JR, and CALL instructions are traced and only the execution of these instructions cause the number of instructions counter to be decremented or the expression (expr)

to be evaluated.

Notice that a conditional RET instruction whose condition is not true (which means the RET instruction is not executed) will not stop program execution but will be traced and will cause the number of instructions counter to be decremented or the expression to be evaulated.

This command will supply the programmer with a history of the past n instructions which altered or could have altered the program counter.

CNJ - TRACE OVER CALLS WITH JUMPS AND NO DISPLAY

```
CNJ
CNJ number of instructions
CNJ {expr}
```

The CNJ command is similar to the CJ command except that no information is displayed as the trace progresses. When the trace terminates, the standard register information is displayed.

D or DM - DISPLAY MEMORY

The contents of memory is displayed in hexadecimal notation. Each line of the display is preceded by the address of the first byte and is followed by the ASCII representation of the hexadecimal bytes. For example:

```
DM100,S30
0100   40 41 42 43  44 45 46 47  48 49 4A 4B  4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0110   50 51 52 53  54 55 56 57  58 59 5A 30  31 32 33 34  PQRSTUVWXYZ01234
0120   35 36 37 38  39 00 00 00  00 00 00 00  00 00 00 00  56789...........
```

If the @ register is not equal to zero the relative address will be displayed as follows (assume the @ register has been set to 100H):

```
DM100,S30
0100 0000' 40 41 42 43  44 45 46 47  48 49 4A 4B  4C 4D 4E 4F  @ABCDEFGHIJKLMNO
0110 0010' 50 51 52 53  54 55 56 57  58 59 5A 30  31 32 33 34  PQRSTUVWXYZ01234
0120 0020' 35 36 37 38  39 00 00 00  00 00 00 00  00 00 00 00  56789...........
```

The formats of this command are as follows:

```
DM
DM beginning-addr
DM beginning-addr ending-addr
DM beginning-addr S swath-width
DM ,ending-addr
DM S swath-width
```

The M is optional in all formats.

The first format displays memory from the current display address, initially 100H, and continues for 8 lines. The second format displays from the beginning address and continues for 8 lines. The third format displays from the beginning address to the ending address. The fourth format displays from the beginning address for a length specified by the swath-width. The fifth format displays from the current display address to the ending address. The sixth format displays from the current display address for a length specified by the swath-width.

DR - DISPLAY REGISTERS

When Debug is re-entered from a break point, the user registers are saved. The registers may be displayed in response to the Debug prompt by typing the following command:

```
-DR
SZHVNCE  A =00 BC =0000 DE =0000 HL =0000 SP=0100 PC=0100 0000' LD   E,A
SZHVNC   A'=00 BC'=0000 DE'=0000 HL'=0000 IX=0000 IY=0000 I=00
```

Notice that the relative address of the program counter (PC) is displayed only if the @ register is not equal to zero.

The letters SZHVNC on the first row represent the flags, while on the second row they represent the prime flags. If the flag is on, it is displayed, if the flag is off, a space is displayed. If only the carry and zero flag are set then " Z C" would be displayed. The flags are described below.

S - Sign flag, S=1 if the MSB of the result is one, i.e., if the result is negative.

Z - Zero flag, Z=1 if the result of an operation is zero.

H - Half-carry flag, H=1 if the add operation produced a carry into the 4th bit of the accumulator or a subtract operation produced a borrow from the 4th bit of the accumulator.

V - Parity or overflow flag. This flag is affected by arithmetic and logical operations. If an overflow occurs during an arithmetic operation, the flag is set to one. After a logical operation, the flag is set to 1 if the result of the operation has even parity.

N - Add/subtract flag, N=1 if the last operation was a subtraction.

C - Carry flag, C=1 if the operation produced a carry.

The E flag on the first line is the state of the interrupt flip-flop (IFF). If interrupts are enabled, the E is displayed, otherwise a space is displayed.

The A register is displayed next, followed by the BC, DE, and HL register pairs and the stack pointer. The program counter value is then displayed as both an absolute and relative address. The opcode pointed to by the program counter is then displayed as an instruction.

On the second line, the prime registers are displayed, F' (prime flags), A', BC', DE', and HL'. The IX, IY, and I (interrrupt page) registers are displayed next. If the disassembled opcode includes an address, the relative value of this address is displayed as the last item on the line.

```
-DR
S H NCE  A =00 BC =0000 DE =0000 HL =0000 SP=0000 PC=1234 0010' CALL 1334
SZ  NC   A'=00 BC'=0000 DE'=0000 HL'=0000 IX=0000 IY=0000 I=00       (0110')
```

E - EXAMINE INPUT PORT

The data port is read and displayed as a hexadecimal number. The format of the command is:

```
E data-port
```

In the following example the data port 3 is read and displayed on the console.

```
-E3
23
```

EJ - EJECT DISK

The format of the command follows:

```
EJ d
```

The d is the disk identifier (A, B, C, etc.). If possible, the diskette in the disk drive will be ejected.

F - SPECIFY FILE NAME

This command allows the programmer to insert file names in the two default FCBs (at 5CH and 6CH) and the command line into the default buffer (at 80H). The example below loads FILE1.COM into the first FCB and FILE2.COM into the second FCB. The complete line is also loaded into the default buffer.

```
-FFILE1.COM FILE2.COM OPTION1 OPTION2
```

This command can be used with the R command to read disk files. This command is compatible with the operation of the CDOS console processor code.

G = GO

The GO command has the following format:

G starting-addr/breakpoint-1 breakpoint-2...breakpoint-n

Each of the addresses is optional. If the starting address is omitted, then the contents of the program counter is used. The registers are loaded from the user registers (these are the values displayed by the DR command). Execution begins with the starting address or the contents of PC (the program counter). If break points were specified, an RST 30H is inserted at the break point addresses and a jump instruction is placed at location 30H. When a breakpoint is executed, control is returned to Debug, and all of the user registers are saved and displayed. All temporary breakpoints (those established by the G command) are then removed from the user program. Note the following about breakpoints:

1. A temporary break point (as used in the G command) may be specified in the same manner and with the same fields (report registers flag, condition, and repeat count) as a permanent break point (see the B command).

2. Breakpoints can only be set in programs residing in RAM. This is because an RST 30H is inserted at each break point location. (The original contents of these locations are saved so that they can be restored after a break point is executed).

3. Up to 12 break points can be set. If an attempt is made to enter more than 12 break points, the command will not be accepted. The 12 breakpoints may be composed of any combination of permanent and temporary breakpoints.

4. When a break point is used, a jump instruction is stored at location 30H. Therefore locations 30H, 31H, and 32H are not available to a user program.

The reader is referred to the B and BX commands for description of permanent and conditional permanent break points.

H = HEXADECIMAL ARITHMETIC

```
H expr
H exprl expr2
```

The first format of the command evaluates the expression and displays the result in hexadecimal and decimal.

The second format will perform hexadecimal addition and subtraction. The first number to be displayed is the sum of the two expressions. The second number is the difference between the first and second expression.

In the following example two hexadecimal numbers (1234H and 321H) are added and subtracted. The first pair of numbers is the result of the addition displayed first in hexadecimal and then in decimal. The second pair of numbers is the result of the subtraction displayed in a similar manner.

```
-H 1234,321
1555 5461.,  0F13 3859.
```

L = LIST IN ASSEMBLER MNEMONICS

The list command is used to list the contents of memory in assembly language mnemonics. The formats of this command are:

```
L
L starting-addr
L starting-addr ending-addr
L starting-addf S swath-width
L ,ending-addr
L S swath-width
```

The first format lists 16 lines of disassembled code starting from the current list address (initially 100H). The second format lists 16 lines from the starting address. The third format lists from the starting address to the ending address. The fourth format lists from the starting address for a length specified by the swath width. The fifth format lists from the current list address to the ending address. The sixth format lists from

the current address for a length specified by the swath-width.

The first address of the disassembly is the absolute address. The second address is the relative address. If the disassembled instruction contains an address, the absolute address is displayed in the instruction in hexadecimal and the relative address is displayed to the right of the disassembled line. In the example that follows, the @ register contains 2800H.

```
-L@800 @811
3000 0800'  ADD  A,B
3001 0801'  CALL 3200      (0A00')
3004 0804'  CALL 3243      (0A43')
3007 0807'  CALL 3333      (0B33')
300A 080A'  LD   A,B
300B 080B'  OR   A,C
300C 080C'  JR   Z,3000    (0800')
300E 080E'  INC  HL
300F 080F'  INC  DE
3010 0810'  INC  BC
3011 0811'  LD   A,H
```

M - MOVE MEMORY

The formats of this command are:

```
M source-addr source-end destination-addr
M source-addr S swath-width destination-addr
```

The first format moves the contents of memory beginning with the source address and ending with the source-end to the destination address. The second format uses the swath width to determine the length of the move.

The move is verified to insure that all bytes were moved correctly. If an overlapping move was made, errors will be reported. The error reporting can be terminated by typing any character.

The move command can be used to fill a block of memory with a constant. In the following example, a zero has been entered into location 100H using the SM command. The following command will move zeros from location 100H through 108H.

```
-M100 S7 101
```

Care should be taken not to move memory over Debug or CDOS.

O = OUTPUT TO DATA PORT

This command outputs data to a data port. The following is the command format:

```
O data-byte port-number
```

P = PROGRAM PROMS

This command enables the user to program PROMs (Programable Read Only Memory chips). The following are the command formats:

```
P source-addr source-end destination-addr
P source-addr S swath-width destination-addr
```

The first format programs PROMs starting with the source address and ending with the source-end into PROMs beginning at the destination address. The second format determines the length from the swath width.

If the length of the source is not a multiple of 400H or if the destination does not begin at a 400H boundary, Debug will reject the command. (Multiples of 400H end in 000H, 400H, 800H, and C00H).

Any number of 2716, 2708 or 2704 PROMs can be programmed in the execution of one command as long as there are enough Cromemco PROM programming boards to contain them. Each PROM is verified with its source after all are programmed and any discrepancies are displayed. If no discrepancies are found, a prompt is displayed and the next command may be entered.

Software can be loaded into a PROM in as small increments as you desire, provided it is added to previously unused areas of the PROM. This is done by first using the Move command (M) to transfer the contents of the PROM to RAM, adding the new

software to an area of RAM which corresponds to the unused portion of the PROM and finally using the Program command (P) to reprogram the PROM with the result. Athough the entire PROM must always be programmed, it never hurts to rewrite the same data over again. In general, a 1 may be written over a 1, a 0 over either a 1 or a 0, but the only way to change 0's to 1's is to erase the PROM with the appropriate UV light. (See the Cromemco Bytesaver II manual for details.)

Q = QUERY MEMORY

The format of the Query command is:

```
Q beginning-addr ending-addr string-of-bytes
Q beginning-addr S swath-width string-of-bytes
```

see last page.

This command is used to search through a specified area of memory for a certain string-of-bytes. The string-of-bytes is in the same format as the S or SM command. If the string-of-bytes is found, 16 bytes starting at the first byte which matches are displayed as in the DM command.

R = READ DISK FILE

The R command is used with the F command to allow the programmer to read a disk file. The F command is used to specify the filename, and the R command causes the file to be read into memory. If the file has a file name extension of HEX, then the file is read as an Intel format HEX file. Any other file is considered to be a binary file and will be read directly into memory beginning at location 100H. The format of the R command is:

```
R
R displacement
```

The first format reads the file with no displacement. The second format reads the file with a displacement. If the input file is in HEX, then the displacement is added to the address in the file to determine the address at which to store the file. If the file is a binary file, it will be stored at the displacement + 100H.

When the R command is executed Debug displays either a question mark if there is an error (file not found, checksum error, or attempting to read a file above highest available memory location) or the following message if there is no error:

```
NEXT = xxxx
NEXTM = yyyy
```

Where xxxx-1 is the highest memory address loaded by all R commands during this Debug session and yyyy-1 is the highest memory address loaded during this R command.

S or SM - SUBSTITUTE MEMORY

This command is used to substitute memory. The formats of this command are:

```
SM
SM starting-addr
```

Note that the M is optional if the first character of the address does not match a register name.

If the first format is used then the last substituted location (initially 100H) will be the starting address.

Debug displays the absolute address, followed by the relative address, followed by the contents of the memory byte. One of the following may then be entered.

1. A data-byte value followed by a carriage return. The data byte value is stored at the address of the prompt. The address is then incremented by 1 and displayed on the next line.

2. A string enclosed between apostrophes (') followed by a carriage return. The string is stored beginning at the address of the prompt. The address is then incremented past the string and displayed on the next line.

3. Any number of 1 and 2 above can be entered on one line with one carriage return terminating the line. The address is then incremented

past the bytes that were stored and the new address is displayed on the next line. .

4. A minus sign (-). A minus sign does not store a byte. The address will be decremented to the previous address. The minus sign can be used to back up to a previous location in case an error was made.

5. A carriage return only. If no entry is made on the line, the memory byte remains unchanged. The address is incremented by 1 and displayed on the next line.

6. A period(.). A period ends the input mode and returns control to the command level.

In the example that follows, assume that the @ register contains the value 2800H:

```
-SM@100
2900 0100' 32 0
2901 0101' 17 00
2902 0102' 31 'this is an ASCII string'
2919 0119' 7A 'AAAA' 0 0 1 2 3 4 5 6 7 8 9
2928 0128' 22
2929 0129' 29
292A 012A' 87 -
2929 0129' 55
292A 012A' 87 .
```

Sr = SUBSTITUTE REGISTER

The Sr command allows the user registers to be altered. The letter r stands for the register which is to be changed. The section SUMMARY OF REGISTER NAMES gives a summary of the names that can be substituted. When substituting the F and F' flags, enter the command SF or SF'. Debug will then display the flags that are set and wait for the programmer to enter the names of the flags which are to be set. If the flags are NOT entered, the flags are reset. Before the following example the SZHC flags are set. After the example the ZC flags are set. The lower case letters are entered by the programmer.

```
-sf
SZH C zc
```

When substituting a one byte register, a one byte value is accepted. When substituting a two byte register, a two byte value is accepted. For example, SD will allow the value of the D register to be changed, SE will allow the value of the E register to be changed, and SDE will allow the value of the D and E registers to be changed.

If no value is entered, or if an error occurs, the value of the register remains unchanged. In the following example, the A register is changed to contain 41H.

```
-sa
A=98 41
```

T = TRACE

The format of the Trace command is:

```
T
T number of instructions
T {expr}
```

The first format traces the program through one instruction. The second format traces the program through number of instructions. The third format traces until the expression (expr) is true (not equal to 0). After every instruction is traced the values of the registers are displayed.

A program can only be traced through RAM. The trace command places a break point after the instruction, loads the registers and executes the instruction. The break point is then executed and the registers are re-saved. The registers are displayed and the next instruction is executed unless the number of instructions counter has reached zero or expr is true, in which case a prompt for the next command is displayed.

To abort the trace, depress any key on the console. A prompt for the next command will then be displayed.

If a permanent break point is reached during tracing, the trace will be stopped.

Examples:

```
T {^B>10}
```

This command will trace the execution of a program, displaying the registers after each instruction, until the B register assumes a value greater than 10H.

TN - TRACE WITH NO DISPLAY

```
TN
TN number of instructions
TN {expr}
```

The TN command is similar to the T command except that no information is displayed as the trace progresses. When the trace terminates, the standard register information is displayed.

TJ - TRACE JUMPS

```
TJ
TJ number of instructions
TJ {expr}
```

The TJ command is similar to the T command except that break points are only placed in the user program before instructions which alter the program counter (i.e., JP, JR, CALL, and RET instructions). The registers will be displayed and the number of instructions counter decremented or the expression evaluated only when a program counter altering instruction is executed. For example:

```
TJ
```

will cause the program to be executed until the next program counter altering instruction is reached. The registers will then be displayed and the programmer prompted for another command.

The registers will be displayed before each PC altering instruction unless the TNJ command is used.

TNJ - TRACE JUMPS WITH NO DISPLAY

```
TNJ
TNJ number of instructions
TNJ {expr}
```

The TNJ command is similar to the TJ command except that no information is displayed as the trace progresses. When the trace terminates, the standard register information is displayed.

V - VERIFY MEMORY

Verify that the block of memory between souce address and source end contain the same value as the block beginning at destination address. The addresses and contents are displayed for each discrepancy found. The following is the format of this command:

```
V source-addr source-end destination-addr
V source-addr S swath-width destination-addr
```

The command works by reading bytes from the source and destination and comparing them.

If a discrepancy is found it is displayed in the following order: source address, source contents, destination contents, destination address. In the example that follows, memory locations 0003H and 1003H do not compare as the same. The same is true of locations 0008H and 1008H.

```
-V 0 S30 1000
0003 32 12 1003
0008 7A 5A 1008
-
```

W - WRITE DISK FILE

The W command is used to write out parts of memory to a disk file. The file name may have previously been specified with the F command or may be the file which was specified on the Debug command line. There are four possible formats:

```
W
W beginning-addr ending-addr
W beginning-addr S swath-width
W ,ending-addr
```

The first format will write out memory from 100H to the last specified by "NEXT =" address (from the last R command). If no R command has been executed, one record (128 bytes) will be written.

In the second and third formats the indicated memory will be written out. The fourth format specifies memory between 100H and ending-addr.

If the number of bytes is not a multiple of 128 (record size), it will be rounded upward. Memory locations less than 100H should not be specified because 5CH contains the FCB and 80H contains the disk buffer.

Z - ZAP MEMORY

```
Z beginning-addr ending-addr string-of-bytes
Z beginning-addr S swath-width string-of-bytes
```

This command is used to initialize a portion of memory. The memory from beginning-addr to ending-addr is initialized with the string-of-bytes repeated over and over. The string-of-bytes is in exactly the same format as in the S or SM command.

Examples:

```
-Z 100 S400 0        (zeros everything from
                      100H to 3FFH)

-Z 1000 S10 0 1 'ABC'
-D 1000 S10
1000  00 01 41 42 43 00 01 41  42 43 00 01 41 42 43 00 ..ABC..ABC..ABC.
```

SUMMARY OF DEBUG COMMANDS

The following is an alphabetical list of the Debug commands.

Command	Description
A	Assemble into memory
B	Set and display Break points
BX	Delete Break points
C	Trace Over Calls
CN	Trace Over Calls with No display
CJ	Trace Over Calls with Jumps
CNJ	Trace Over Calls with Jumps and No Display
D or DM	Display Memory
DR	Display Register
E	Examine input port
EJ	EJect disk
F	Specify disk File name
G	Go
H	Hexadecimal arithmetic
L	List in assembler mnemonics
M	Move memory
O	Output to data port
P	Program PROMs
Q	Query memory
R	Read disk file
S or SM	Substitute Memory
Sr	Substitute register (refer to Summary of Register Names)

T	Trace
TN	Trace with No display
TJ	Trace Jumps
TNJ	Trace Jumps with No Display
V	Verify memory
W	Write disk file
Z	Zap memory

SUMMARY OF REGISTER NAMES

The following register names are displayed by the DM command and may be used with the Sr command.

Register Decription

F Flags, the following flags may be changed.

S -Sign flag
Z -Zero flag
H -Half carry flag
V -parity/oVerflow flag
N -subtractioN flag
C -Carry flag

The interrupt enable flag (E) may also be changed.

F' The F' flags are the same as the F flags. (Note that the E flag may not be changed here.)

A accumulator

A' Prime Accumulator

B B register

B' B' register

C C register

C' C' register

D D register

D' D' register

E	E register
E'	E' register
H	H register
H'	H' register
L	L register
L'	L' register
BC	BC register pair
BC'	BC' register pair
DE	DE register pair
DE'	DE' register pair
HL	HL register pair
HL'	HL' register pair
SP S	Stack Pointer
PC P	Program Counter
IX X	IX register
IY Y	IY register
I	Interrupt page register
IP	Interrupt page register as top byte

Note:

The disk file directory is now available to the user. In response to the CDOS prompt type:

```
Debug SYS.DIR
```

This will load the disk directory into memory starting at location 100H. The programmer may examine it (using the DM command), change it (using the SM command), and if necessary write it back to the disk (using the W command). Users who are not familiar with the structure of the disk directory should not attempt to use this feature.

I N D E X

Q begin address end address 'string' (string)

Q begin address end address AB FF (hex numbers)

Q begin address end address 58. (numbers)