ROM2/Cleuseau Model 100/102
Developers Version
Supplement

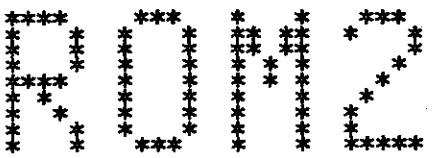
Note: This document supplements the regular ROM2/Cleuseau ROM manual. The Developers version of this ROM is available from Club 100 for Modes 100/102 only — a developers version is not available for the Model 200 or NEC PC8201a.

Cleuseau

Text Editor and BASIC Inspector

User's Manual

• TRS-80 Model 100 • Tandy 200 • Nec PC-8201A •



for
the Model 100™
or
the TANDY 200™
or
the NEC 8201A™

User's Manual for ROM2 v5.1

Club 100: The Model "T" User Group (since 1983)
P.O. Box 23438, Pleasant Hill, CA 94523-0438
925-932-8956, fax 937-5039, bbs 939-1246
www.the-dock.com/club100.html

ROM2/Cleuseau Model 100/102 Developers Version Supplement

Note: This document supplements the regular ROM2/Cleuseau ROM manual. The Developers version of this ROM is available from Club 100 for Modes 100/102 only — a developers version is not available for the Model 200 or NEC PC8201a.

Deleted CMD modes:

CALL 913,145
No longer activates Cleuseau.

CALL 913,193
No longer calculates ROM2's 8K checksum.

CALL 911'command str
No longer supported.

ROM2 entry points
ROM2 machine code entry points no longer supported.

New Cleuseau command:

ctrl-R
Invokes ROM2 (if Cleuseau is active).

Model 100/102 new CMD modes:

RC$="str":CALL 911
Invokes ROM2 with value of RC$ as the command input.

CALL 913,17
Invokes DBG directly. Useful for invoking DBG directly from machine code.

CALL 913,129
Load machine code from TDD via routine.

CALL 63013,1
Enables Cleuseau.

CALL 63013,144
Disables Cleuseau.

Tandy 200 new CMD modes:

RC$="str":CALL 921,2

Invokes ROM2 with value of RC$ as the command input.

CALL 921,18
Invokes DBG directly. Useful for invoking DBG directly from machine code.

CALL 61167,2
Enables Cleuseau.

CALL 61167,144
Disables Cleuseau.

New messages:

Attach TDD!
This message is displayed if a TDD is not connected.

TDD err
This message is displayed if a TDD command cannot be completed properly.

?FC error
If ROM2 fails to complete a command as instructed, a ?FC error is issued.

New CMD commands:

RUN []
Load from the TDD and run it. (.BA and .DO files are loaded into RAM as while .CO files are loaded in the HIMEM-MAXRAM area. If is omitted then the name is used.

LOAD []
Load from the TDD into RAM as . If is omitted then the name is used.

SAVE []
Save in RAM to the TDD as . If is omitted then the name is used.

KILL
Kill on the TDD. (No confirmation.)

FILES
List TDD files and sizes.

FORMAT
Format the TDD disk. (No confirmation.)

TDD file types:

ROM2 uses the two character extension in TDD file names for determining the file type.

ROM2 allows these extensions: .DO, .CO, .BA, and .BR.

Extensions in ROM2 commands which have two file names must have equivalent file types. If the extension is omitted in the second file name then the extension of the first is used. If the extension is omitted in the first file name then .DO is used.

Strong and weak machine file types:

ROM2 allows a third character in the TDD file extension. Weak machine file type checking (the default mode)

allows any character. Strong checking allows only the machine's corresponding digit (if the third character is blank then the machine's digit is added). Machine file type checking helps when the TDD files are a mixture of Model 100, NEC 8201A, and/or Tandy 200 files. Since .CO files are almost never compatible, running the wrong machine's file will probably lead to a machine crash and data loss. Also, NEC and Tandy BASIC programs are tokenized differently and will probably corrupt RAM if loaded into the wrong machine type. Strong machine file type checking prevents loading the wrong type of file into the machine.

Weak machine file type mode:

100: POKE 63852,0
Third character in the file extension can be anything.

200: POKE 62044,0
Third character in the file extension can be anything.

NEC: POKE 63577,0
Third character in the file extension can be anything. Example (assuming the machine is a Model 100):

SAVE PROG.BA PROG.BA1
Save RAM file PROG.BA as TDD file PROG.BA1.

SAVE PROG.BA PROG.BA2
Save RAM file PROG.BA as TDD file PROG.BA2.

Strong machine file type mode:

100: POKE 63852,32
Third character in the file extension must be '1'.

200: POKE 62044,32
Third character in the file extension must be '2'.

NEC: POKE 63577,32
Third character in the file extension must be '0'.

If the third char is blank then the proper machine char is added. Example (assuming the machine is a Model 100):

SAVE PROG.BA PROG.BA1
Save RAM file PROG.BA as TDD file PROG.BA1.

SAVE PROG.BA PROG.BA2
Results in a ?FC error, due to machine type mismatch.

Example ROM2/TDD program:

```
10 INPUT "File";F$
20 IF F$="" THEN MENU
30 RC$="LOAD "+F$:CALL 911
40 RC$="ASMN "+F$+" "+F$:CALL 911
50 RC$="SAVE "+F$+".CO":CALL 911
60 MENU
```

****** ASM/DBG Modifications ******

New ASM modes:

ASM []
Assembles .DO to .CO If is omitted then machine code is placed directly into the HIMEM-MAXRAM area.

LAB.DO
An _ASM_.DO file can be renamed _LAB_.DO and used as the initial symbol table for ASM. This reduces the time and memory required to assemble programs which reference many constant label values.

New ASM pseudo ops:

ENTRY
Define the current PC ($) as the EXEcute address for the .CO file created by ASM.

DEC
Untyped constants are converted to decimal. This is the initial mode of ASM.

HEX
Untyped constants are converted to hexadecimal.

Extended ASM pseudo ops:

&file
&&file
If file.DO is not in RAM then load it from the TDD, include it and then delete it. Macros can not be defined in a file loaded this way.

New ASM/DBG constant types:

[0..9]* 'T'
Decimal constant. This constant type is needed for entering decimal constants while in HEX mode, since 'D' is a valid hex digit.

[0..1]* 'Y'
Binary constant. This constant type is needed for entering binary constants while in hex mode, since 'B' is a valid hex digit.

New DBG commands:

F [],[]
Find all memory blocks matching the one that starts at address and is long. If is omitted then a length of three (3) is used. If is omitted then the current PC is used.

DEC
Display register values in decimal. Untyped constants are converted to decimal.

HEX
Display register values in hexadecimal. Untyped constants are converted to hexadecimal. This is the initial mode of DBG.

LPT
All DBG display goes to the printer, including prompts and keyboard entry.

COM
All DBG display and input via the COM/MDM port. Use TELCOM's STAT command or a BASIC OPEN statement to set the communication parameters.

EQ [],[]
Breakpoint if the byte at is equal to . If is omitted then the current value at is used. If is omitted then the current

PC is used.

NEQ [],[]
Breakpoint if the byte at is not equal to . If is omitted then the current value at is used. If is omitted then the current PC is used.

CST
Clear the stack trap address. DBG automatically exits when the RET at the end of a routine that invoked DBG is simulated. CST turns off the auto exit.

D
Display registers and the next instruction to be simulated. All values displayed in decimal or hexadecimal, controlled by the DEC and HEX commands. DBG starts in HEX mode. Values and addresses displayed as the equivalent label name if possible.

L []
Disassemble opcodes starting at . All values displayed in decimal or hexadecimal, controlled by the DEC and HEX commands. DBG starts in HEX mode. Values and addresses displayed as the equivalent label name if possible.

N
as first debug command. Causes subsequent L commands to disassemble the Cleuseau/ROM2 addresses. This allows you to examine and test the contents of the Cleuseau/ ROM2. Also can run/simulate ROM2 routines.

******Other Notes ******

RAM
ROM2/Cleuseau uses the following areas of RAM:

```
ROM_Check:   T200:EEC8..EEEE
IN_Buf:      T200:EFC0..F05F
ROM_Name:    T200:F4D3..F4DA
Page2:       T200:F7C8..F8CF
```

ROM2 uses the last two-thirds of In_buf for operating on user input. This limits the usable input line length to 80 chars while in DBG or other ROM2 functions.

In the T200, the Page2 area starts at F7B0 and runs for 640 bytes to FA2F. ROM2 and Cleuseau use the F7C8..F8CF part only.

The Original Laptop Computer . . . 1983

Cleuseau

Text Editor and BASIC Inspector

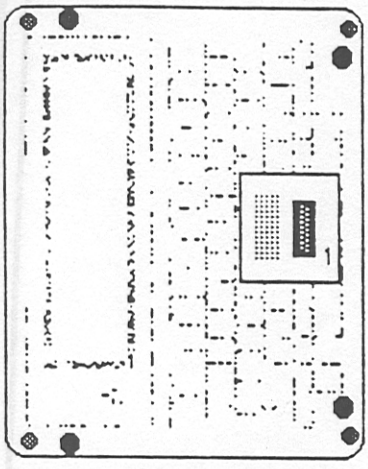
User's Manual

• TRS-80 Model 100 • Tandy 200 • Nec PC-8201A •

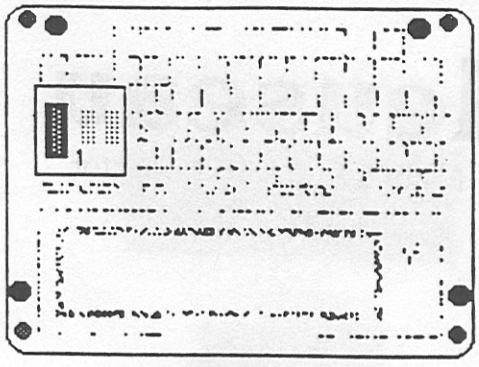
July 1, 1985
(second printing)

Polar Engineering and Consulting
P.O. Box 7188 Nikiski, Alaska 99635
(907) 776-5529

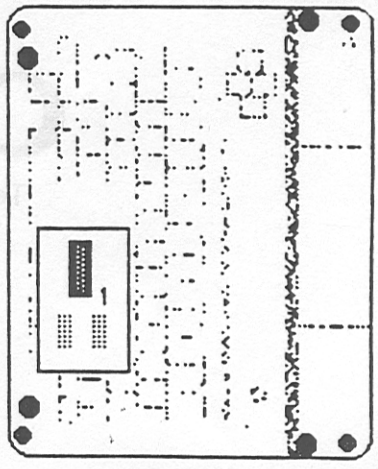
How to insert the option ROM into its socket.



model 100
bottom view



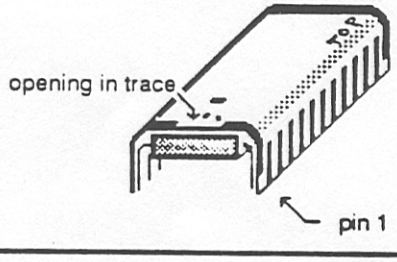
model 102
bottom view



model 200
bottom view

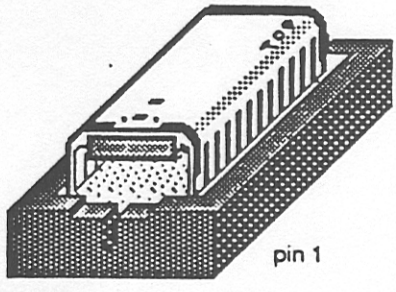
1 Turn off the power to your computer. (It is not necessary to shut down memory power.) Turn the computer upside down on a soft surface, oriented as shown, depending on whether you have a model 100, 102 or 200. Use a dime or a screwdriver to snap off the protective plastic cover from the options compartment. Locate the option ROM socket and pin 1 on that socket. Pin 1 will be at the corner illustrated. Also, you will find a numeral "1" printed on the circuit board by the corner.

2 Locate pin 1 on the eprom carrier.

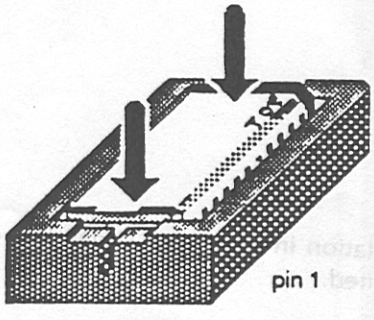


power off?!

3 Line up eprom carrier pin 1 with socket pin 1, and set the carrier gently on top of the opening in the socket.



4 Press straight down firmly and evenly with two thumbs, until the carrier is seated as deep as it can go inside the socket.



5 Replace the cover on the options compartment, and proceed to test your new option ROM.

To remove the carrier, pry up gently at the two ends.

Club 100 ROMs are constructed as a one-piece unit. DO NOT SEPARATE COMPONENTS. Lay ribbon (included) into ROM socket, first, for easy removal. Yes, it's a tight fit. Call if you have ANY questions.

CONTENTS

IMPORTANT

Polar Engineering and Consulting sells Cleuseau for the Model 100, Tandy 200 and NEC 8201A. Though the three versions of Cleuseau are very much alike, they are machine specific.

A Model 100 Cleuseau will not work with an NEC 8201A. Likewise, an NEC 8201A Cleuseau will not work with a Model 100. A Tandy 200 Cleuseau will not work with a Model 100. etc.

This manual explains how to use Cleuseau on all machines. Variations between the Model 100, Tandy 200 and NEC 8201A are noted.

WARNING

Do NOT remove the Cleuseau ROM or any other option ROM with the computer's main power on. This could the damage the ROM and corrupt RAM memory. Always turn off the main power before installing or removing Cleuseau. (It is NOT necessary to turn off the RAM memory power.)

WARNING (NEC 8201A only)

Always DISABLE Cleuseau before loading in either the CRT or DISK drive support software.

HARDWARE INSTALLATION (Model 100/200)

WARNING: Improper installation may erase RAM memory or damage Cleuseau or the Model 100/200. Cleuseau is sensitive to static electricity so exercise caution and FOLLOW installation instructions. READ STEPS 1 THROUGH 10 BEFORE STARTING.

Handling: Do not touch Cleuseau's pins or the metal contacts on the circuit board. DO NOT REMOVE THE CIRCUIT BOARD; it is essential to Cleuseau's functionality.

Step 1: Place a soft towel on a flat table surface.

Step 2: Turn the Model 100/200 OFF and place it upside down on the towel. Orient the Model 100/200 with the removable plastic panel nearest you.

Step 3: Insert a penny into the slot on the near side of the panel and pry the panel off.

Step 4: Correctly orient Cleuseau with its label reading from left to right. **LOOSELY** place Cleuseau into the option ROM socket (M11, it is the nearer of the two). DO NOT USE FORCE IN THIS STEP!

Step 5: Cleuseau's label should read normally. If **NOT** then go back to the previous step and repeat it.

Step 6: Press Cleuseau evenly and steadily with your thumbs into the socket until it will not go down any more.

Step 7: Replace the panel.

Step 8: Turn the Model 100/200 face up ready for typing.

Step 9: Turn on the power. If the main menu does not appear, turn off the power immediately and remove Cleuseau and repeat entire installation from the first step.

Step 10: If the main menu did appear then the hardware installation is done. Proceed to the software installation.

HARDWARE INSTALLATION (NEC 8201A)

WARNING: Improper installation may erase RAM memory or damage Cleuseau or the NEC 8201A. Cleuseau is sensitive to static electricity so exercise caution and FOLLOW installation instructions. READ STEPS 1 THROUGH 10 BEFORE STARTING.

Handling: Do not touch Cleuseau's pins. Three pins on Cleuseau may have been modified. This is correct and necessary. DO NOT ATTEMPT TO ALTER Cleuseau IN ANY WAY!

Step 1: Place a soft towel on a flat table surface.

Step 2: Turn the NEC 8201A off and place it upside down on the towel. Orient the NEC 8201A with the removable plastic panel nearest you.

Step 3: Remove the three philips screws and the plastic cover.

Step 4: Correctly orient Cleuseau with its label reading from left to right. Rotate it 90 degrees counter-clockwise. Place it on the second socket from the left. (The first socket has the MicroSoft ROM in it.)

Step 5: Cleuseau's label should read the same way as the MicroSoft ROM. If **NOT** then go back to the previous step and repeat it.

Step 6: Using both thumbs, press Cleuseau firmly into its socket. Make sure that all the pins are inserted.

Step 7: Replace the panel.

Step 8: Turn the NEC 8201A face up ready for typing.

Step 9: Turn on the power. If the main menu does not appear, turn off the power immediately and remove Cleuseau and repeat entire installation from the first step.

Step 10: If the main menu did appear then the hardware installation is done. Proceed to the software installation.

SOFTWARE INSTALLATION

Cleuseau software is installed from BASIC. Enter from BASIC:

Model 100: **CALL 63012**
Tandy 200: **CALL 61167,2**
NEC 8201A: **EXEC 62393**

This only needs to be done once. When installed and enabled the word "Cleuseau" will appear on the main menu. If, after entering this command, the menu does not automatically appear then Cleuseau may not be installed correctly. Repeat the hardware installation.

If your computer has multiple RAM banks you will have to install the Cleuseau software in each bank you wish to use.

Cleuseau remains installed until a COLD-RESTART occurs or you disable it. To disabled Cleuseau, enter from BASIC:

Model 100: **CALL 913,145**
Tandy 200: **CALL 921,146**
NEC 8201A: **EXEC 1124,145**

If you physically remove Cleuseau from your computer, Cleuseau will automatically disable itself. You will have to repeat the installation instructions completely to put Cleuseau back in your computer. (Note: After removal, the word "Cleuseau" which appears on the main menu may not disappear until a menu choice has been made.)

AN OVERVIEW OF CLEUSEAU ENHANCEMENTS TO MICROSOFT TEXT

Filing (Model 100/200 only)

Save.....(Model 100/200 only) Prompts for confirmation when saving null files. This helps protect disk files when **Save** is entered by mistake.

CTRL_V...(Model 100/200 only) Compare the file with a saved file. Cursor advances to the first non-matching char.

Mode Setting

CTRL_J...Toggle Cleuseau TEXT on/off.

CTRL_O...Overwrite mode.

Moving

CTRL_G...Go to the next line following the next carriage return.

Character Editing

CTRL_P...Insert a control character. (New for the Nec 8201A.)

Searching and Replacing

Ffnd.....Fixed the MS TEXT search bug. Also, allows **CTRL_P CTRL_<char>** in the search string.

Bfnd.....Search backwards to the previous occurrence of a string. Allows the search for control chars, like **Ffnd**.

CTRL_N...Search for the next occurrence of the search string.

CTRL_S...Replace and search. If the cursor is at the search string, replace it with the replacement string. Search for the next occurrence of the search string. Allows control characters in either strings.

CTRL_Y...Repeat replace and search with current strings.

Selecting and Pasting

Copy.....Allows selection of text AFTER the command.

Cut.....Allows selection of text AFTER the command.

CTRL_E...Append to the paste buffer. Copy text from the file and add it to the end of the paste buffer.

CTRL_L...Convert text to lowercase.

CTRL_U...Convert text to uppercase.

Miscellaneous

ESC.....Repeat command. Any character, digit or symbol key may be repeated using **ESC**. Also, TEXT functions **Ffnd**, **Bfnd**, **PASTE**, **BKSP**, **DEL**, **TAB**, **ENTER**, **CTRL_D**, **CTRL_G**, **CTRL_N**, **CTRL_P**, **CTRL_S**, **CTRL_Y**, arrow keys and **SHIFT** arrow keys may be repeated.

CTRL_@...Displays the current cursor location.

CTRL_D...Displays the number of bytes, number of words and number of lines between the top of the file and the cursor.

CTRL_X...HELP text for all Cleuseau TEXT commands.

Command Key Descriptions

All command keys used by Cleuseau are described here. Many keys are used the same as defined in the Model 100 User's Manual (pages 43-60), the Tandy 200 Owner's Manual (Chapters 2,3) and the NEC PC-8201A User's Guide (Chapter 7). Page references are given for those keys. Cleuseau alters the function of some keys. Those keys are described in detail.

After some descriptions are one or more examples of appropriate commands. If a command requires more than one keystroke, Cleuseau will prompt for an appropriate response.

Experiment with Cleuseau's commands. Most are easy to understand when you see them in action.

Filing

Load .....(Model 100/200 only) Same as MS TEXT (M100 pg 43, T200 pg 11). Load file from CAS:, COM:, MDM:, 0:, or 1:. Prompts for the file to be loaded.

Save .....(Model 100/200 only) Same as MS TEXT (M100 pg 43, T200 pg 10) except null files require confirmation before being saved. (This prevents accidentally saving a null file on DISK when loading was intended. The keys are next to each other.) Save file to CAS:, COM:, MDM:, 0:, or 1:. Prompts for the file to be saved.

CTRL V .....(Model 100/200 only) Compare the file with one that has been saved, to Verify that it is the same. Cursor is advanced from the top of the file to the first non-matching character. Very useful for verifying a save to cassette tape. Verify file with CAS:, COM:, MDM:, 0:, or 1:.

Mode Setting

LABEL/Keys ..When Cleuseau is not enabled (see **CTRL J**), the functions of the eight keys are shown, as MS TEXT uses them. When Cleuseau is enabled, the functions of the eight keys are shown, as Cleuseau uses them.

LABEL (Model 100/200)

Ffnd	Load	Save	Bfnd	Copy	Cut	Sel	Menu
F1	F2	F3	F4	F5	F6	F7	F8

Keys (NEC 8201A)

Ffnd	Next	Sel	Cut	Copy	Bfnd	Keys	Menu
F1	F2	F3	F4	F5	F6	F7	F10

CTRL J .....Toggle switch. Turn off all Cleuseau TEXT commands leaving the computer with only its original MS TEXT commands, or restore Cleuseau TEXT commands.

CTRL O .....Toggle between INSERT and OVERWRITE modes. MS TEXT has only INSERT mode. In OVERWRITE mode:

Non-control chars (normal text) overwrite the current cursor and advance the cursor.
Control chars and carriage returns are still inserted.
Ends of lines are stretched.
BKSP blanks out previous char if it is a non-control char and moves back one space, leaving a blank space.
BKSP deletes control chars and carriage returns.
DEL deletes the char under the cursor in both INSERT and OVERWRITE modes, leaving no blank. Same as MS TEXT (M100 pg 7, T200 pg 8, NEC pg 7-10).
Paste buffer is always inserted.

A message showing the mode is printed when the mode is changed.

Moving

←, **→**, **↑**, **↓** ....Same as MS TEXT (M100 pg 45, T200 pg 12, NEC pg 7-9).

SHIFT ←, **→**, **↑**, **↓** ..Same as MS TEXT (M100 pg 45, T200 pg 14, NEC pg 7-9).

CTRL ←, **→**, **↑**, **↓** ..Same as MS TEXT (M100 pg 45, T200 pg 14, NEC pg 7-9).

CTRL G .....Go to the next line after the next carriage return.

Character Editing

BKSP .....Same as MS TEXT (M100 pg 7, T200 pg 8, NEC pg 7-10) except in overwrite mode (see **CTRL_O**). Erase char to the left of the cursor. Same as **CTRL_H**.

DEL .....(**SHIFT_BKSP**) Same as MS TEXT (M100 pg 7, T200 pg 9, NEC pg 7-10). Delete char under cursor.

ENTER .....Same as MS TEXT (M100 pg 7, T200 pg 8, NEC pg 7-28). Insert a carriage return and a line feed. Same as **CTRL_M**.

TAB .....Same as MS TEXT. Move cursor to next tab position by inserting a **TAB** character. Same as **CTRL_I**.

CTRL P .....Same as MS TEXT (M100 pg 60, T200 not documented but it works, new for the NEC). Insert a control char into the file.

CTRL_P CTRL_L

Insert a form feed printer command into the file.

Searching and Replacing

Ffnd .....Search forwards. Prompts for a search string. (Defaults to the last search string used.) Advances to the next occurrence of the search string. Search starts at the char following the cursor. (Cleuseau fixes the MicroSoft bug that causes a string to be "not found" if the string is preceded with a duplicate of the first character. For example: "ok" would be missed in the word "book".) Control chars may be searched for.

Ffnd C H E E R S ENTER

Searches for the next occurrence of "CHEERS".

Ffnd CTRL_P CTRL_L ENTER

Searches for the next occurrence of a form-feed.

Note: ^ appears as ^^ to distinguish it from ^**char**, which is a control character.

Bfnd .....Like **Ffnd**, except the search proceeds backwards from the cursor to the beginning of the file. Search starts at the char preceding the cursor. Prompts for a search string. Control chars may be searched for. (See **Ffnd**.)

CTRL N .....Search for the **Next** occurrence of the search string. Search in the same direction as the most recent **Ffnd** or **Bfnd** command. (NEC 8201A: same as **Next**.)

CTRL S .....Replace and Search. Prompts for a search string. (Defaults to the last search string used.) Prompts for a replacement string. (Defaults to the last replacement string used.) If search string is found at the current cursor position then it is replaced with the replacement string. In any case, this command searches forwards for the next occurrence of the search string. (Same as **PAUSE**).

Use **CTRL_Y** to repeat the same replace and search again without having to retype the strings.
Use **CTRL_N** to repeat the same search but without replacing that particular occurrence of the search string. (See **CTRL_Y** and **CTRL_N**.)

CTRL Y .....Repeat the replace and search (see **CTRL_S**) with the current search and replacement strings.

Selecting and Pasting

Sel .....Same as MS TEXT (M100 pg 44, T200 pg 15, NEC pg 7-18). Set select marker at cursor position. To select text, this must be followed by one or more move commands, not necessarily the cursor arrows.

Sel SHIFT_→ SHIFT_→

Selects the next two words as those to be used.

Sel is used with command keys **Copy**, **Cut**, **CTRL_E**, **CTRL_L**, and **CTRL_U**.

Copy .....Same as MS TEXT (M100 pg 44, T200 pg 16, NEC pg 7-22). Copy selection into paste buffer, erasing whatever was previously there. This does not remove the selection from the file. See "Advanced Usage" for special **Copy** Cleuseau commands.

Cut .....Same as MS TEXT (M100 pg 44, T200 pg 15, NEC pg 7-20). Cut selection into paste buffer, erasing whatever was previously there. This removes the selection from the file. See "Advanced Usage" for special **Cut** Cleuseau commands.

CTRL E .....Append to the paste buffer. Same as **Copy** except the selected chars are added to the **End** of the paste buffer instead of replacing the contents of the paste buffer. See "Advanced Usage" for special **CTRL_E** commands.

CTRL L .....Change the selected chars to **L**owercase. See "Advanced Usage" for special **CTRL_L** commands.

CTRL U .....Change the selected chars to **U**ppercase. See Advanced Usage for special **CTRL_U** commands.

Sel CTRL_→ CTRL_U

Select the line to the right of the cursor and capitalize it.

PASTE .....Same as MS TEXT (M100 pg 44, T200 pg 16, NEC pg 7-26). Insert a copy of the paste buffer at the cursor position.

Miscellaneous

BREAK/STOP ..(SHIFT_PAUSE) Same as MS TEXT (M100 pg 44, T200 pg 12, NEC pg 3-6). Abort any operation. This key will terminate ESC repeat commands.

ESC .....Repeat function. See Cleuseau Advanced Usage.

ESC 8 ENTER SHIFT_→

Move eight words forwards.

Sel ESC 1 0 ENTER → CTRL_U

Capitalize next ten characters.

Menu .....Same as MS TEXT (M100 pg 44, T200 pg 9, NEC pg 7-25). Return to the main Menu or BASIC.

PRINT .....(Model 100/200 only) Same as MS TEXT (M100 pg 44, T200 pg 6). Print screen contents on the printer.

SHIFT PRINT ..(Model 100/200 only) Same as MS TEXT (M100 pg 44, T200 pg 9). Print entire text file on the printer.

CTRL @ .....Display the current cursor position. (Row #, Column #)

CTRL D .....Display the file status. Print a message showing the number of bytes, number of words, and number of lines (carriage returns) between the top of the file and the cursor. (Note: When EDITing a BASIC program, this command displays the number of bytes in the ASCII form.)

CTRL X .....HELP. Quick reference to all of the command keys.

CTRL_X CTRL_X

Whole screen shows a list of all command keys and description of each. To return to editing, hit any key except ENTER.

CTRL_X CTRL_G

Bottom line of screen shows a description of the key requested, in this case: "CTRL_G go to next line".

The use of ALL command keys is described here and in the printed Quick Reference.

These command keys are duplicates of others. They are features of the Model 100, Tandy 200 and NEC 8201A software and are not covered in depth here.

CTRL A .....Same as **SHIFT_←**. Move cursor to the beginning of the word to the left. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL B .....Same as **SHIFT_↓**. Move cursor to the Bottom of the display in the current column. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL C .....Same as **BREAK/STOP**. Cancel any command. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 3-6).

CTRL F .....Same as **SHIFT_→**. Move cursor to the beginning of the next word. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL H .....Same as **BKSP**. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL I .....Same as **TAB**. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL K .....Not available to the user.

CTRL M .....Same as **ENTER**. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-28).

CTRL Q .....Same as **CTRL_←**. Move cursor to the left end of the current line. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL R .....Same as **CTRL_→**. Move cursor to the Right end of the current line. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL T .....Same as **SHIFT_↑**. Move cursor to the Top of the display in the current column. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL W .....Same as **CTRL_↑**. Move cursor to the beginning of the file. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

CTRL Z .....Same as **CTRL_↓**. Move cursor to the end of the file. MS TEXT (M100 pg 60, T200 pg 54, NEC pg 7-9).

Advanced Usage

This section explains special combinations of command keys. The Advanced Usage is intended for the speed and convenience of the user.

Post Selection of Text:

Cleuseau allows an alternate use of Copy, Cut, CTRL_K, CTRL_L and CTRL_U. If no chars have been selected (by Sel <move>s) before one of these five commands is used, a single <move> key or ESC n ENTER <move> is used to define the range of chars.

Some commands are NOT repeatable using the ESC n syntax, including Copy, Cut, CTRL_K, CTRL_L and CTRL_U. For those commands ESC may precede the <move> part only. If ESC n immediately precedes a command that cannot be repeated, the message "aborted" will appear on the screen.

Copy ..... Copy <move>

Move as directed. Copy to the paste buffer all chars the cursor has moved past.
Same as Sel <move> Copy.

Copy ESC n ENTER <move>

Repeat the move n times. Copy to the paste buffer all chars the cursor has moved past.
Same as Sel ESC n ENTER <move> Copy.

Cut ..... Cut <move>

Move as directed. Cut to the paste buffer all chars the cursor has moved past. Same as Sel <move> Cut.

Cut ESC n ENTER <move>

Repeat the move n times. Cut to the paste buffer all chars the cursor has moved past.
Same as Sel ESC n ENTER <move> Cut.

CTRL E ..... **CTRL_E <move>**

Move as directed. Copy the chars the cursor has moved past and add them to the end of whatever is in the paste buffer.
Same as **Sel <move> CTRL_E.**

CTRL_E ESC n ENTER <move>

Repeat the move **n** times. Copy the chars the cursor has moved past and add them to the end of whatever is in the paste buffer.
Same as **Sel ESC n ENTER <move> CTRL_E.**

CTRL L ..... **CTRL_L <move>**

Move as directed. Lowercase all chars the cursor has moved past. Same as **Sel <move> CTRL_L.**

CTRL_L ESC n ENTER <move>

Repeat the move **n** times. Lowercase all chars the cursor has moved past.
Same as **Sel ESC n ENTER <move> CTRL_L.**

CTRL U ..... **CTRL_U <move>**

Move as directed. Capitalize all chars the cursor has moved past. Same as **Sel <move> CTRL_U.**

CTRL_U ESC n ENTER <move>

Repeat the move **n** times. Capitalize all chars the cursor has moved past.
Same as **Sel ESC n ENTER <move> CTRL_U.**

ESC-Repeat Function:

An example of proper use of the **ESC** key in each instance is followed by a description of the command results.

Note: **ESC R ENTER** can be used to make the next command repeat, an indefinite number of times. This is most useful when used with search commands: **Ffnd, Bfnd, CTRL_N, CTRL_S, CTRL_Y.** For example:

ESC R ENTER CTRL_S <string> ENTER <string> ENTER

Replaces all occurrences of the first **<string>** with the second **<string>**. (Starting at the cursor and moving in the direction of the previous search.)

Ffnd ..... **ESC n ENTER Ffnd <string> ENTER**

Finds the nth string, searching forwards from the cursor. If less than **n** strings are found, the cursor will not move and a message shows how many occurrences of the string were found.

ESC n ENTER Ffnd CTRL_P CTRL_Z ENTER

Moves forward **n** bytes. (This is a useful artifact.)

Bfnd ..... **ESC n ENTER Bfnd <string> ENTER**

Functions the same as **Ffnd**, except the search proceeds backwards from the cursor.

PASTE ..... **ESC n ENTER PASTE**

At the current cursor position, insert **n** copies of the paste buffer.

ENTER ..... **ESC n ENTER ENTER**

Same as typing **ENTER**, **n** times.

TAB ..... **ESC n ENTER TAB**

Same as typing **TAB**, **n** times.

BKSP ..... ESC n ENTER BKSP

Same as typing BKSP, n times.

<char> ..... ESC n ENTER <char>

Same as typing <char>, n times.

<char> refers to a single character, digit, symbol or graph char. Usage for control chars, function keys, command keys or cursor keys is described elsewhere.

<move> ..... ESC n ENTER <move>

Same as typing <move>, n times.
Valid <move> keys:

←	SHIFT_←	Ffnd
→	SHIFT_→	Bfnd
↑	SHIFT_↑	CTRL_G
↓	SHIFT_↓	CTRL_N

CTRL_P ..... ESC n ENTER CTRL_P <CTRL_char>

Inserts n control chars <CTRL_char> into the file.

CTRL_S ..... ESC n ENTER CTRL_S <string> ENTER <string> ENTER

Replaces the search string (first <string>) with the replacement string (second <string>). If the cursor is at a search string, the next n occurrences are replaced. Otherwise, the next n-1 occurrences are replaced.

If fewer than n (or n-1) are found, a message will show how many are found and replaced. Otherwise, the cursor advances to the next occurrence that is not replaced.

CTRL_Y ..... ESC n ENTER CTRL_Y

Replaces the current search string with the current replacement string, n times. (See CTRL_S)

CLEUSEAU TEXT

FILING (Model 100/200 only)

Load ..... Load from file
Save ..... Save to file
CTRL V..... Verify file with saved file

MODE SETTING

LABEL/Keys .... Toggle label line on/off
CTRL J..... Toggle Cleuseau TEXT on/off
CTRL O ..... Toggle INSERT/OVERWRITE

CHARACTER EDITING

BKSP ..... Delete char left of cursor CTRL H
DEL..... Delete char under cursor
ENTER ..... Insert CR and line feed
TAB..... Insert tab to column 1,9,17,..CTRL I
CTRL P..... Insert control character

MISCELLANEOUS

BREAK/STOP ... Cancel any command
ESC..... n times, repeat command
Menu..... MENU or BASIC (exit TEXT)
PRINT..... Print screen on printer
SHIFT PRINT ... Print file on printer
CTRL @ ..... Display cursor location
CTRL D..... Display number of bytes/words/lines
CTRL X..... HELP (CTRL X CTRL X for all)

MODEL 100/200 FUNCTION KEYS

Ffnd	Load	Save	Bfnd	Copy	Cut	Sel	Menu
F1	F2	F3	F4	F5	F6	F7	F8

CLEUSEAU TEXT

MOVING

← ..... Move left one char
SHIFT ← ..... Move to start of word (left) CTRL A
CTRL ← ..... Move to end of line (left) CTRL Q
→ ..... Move right one char
SHIFT → ..... Move to start of word (right) CTRL F
CTRL → ..... Move to end of line (right) CTRL R
↑ ..... Move up one line
SHIFT ↑ ..... Move to top of screen CTRL T
CTRL ↑ ..... Move to top of file CTRL W
↓ ..... Move down one line
SHIFT ↓ ..... Move to bottom of screen CTRL B
CTRL ↓ ..... Move to end of file CTRL Z
CTRL G ..... Go to the next line

SEARCHING AND REPLACING

Ffnd ..... Search forwards
Bfnd ..... Search backwards
CTRL N ..... Search for next string
CTRL S ..... Replace and search (PAUSE)
CTRL Y ..... Repeat replace and search

SELECTING AND PASTING

Sel ..... Set select marker at cursor
Copy ..... Copy selection to paste buffer
Cut ..... Cut selection to paste buffer
CTRL E ..... Append selection to paste buffer
CTRL L ..... Lowercase selection
CTRL U ..... Uppercase selection
PASTE ..... Insert copy of paste buffer

NEC 8201A FUNCTION KEYS

Ffnd	Next	Sel	Cut	Copy	Bfnd	Keys	Menu
F1	F2	F3	F4	F5	F6	F7	F10

AN OVERVIEW OF CLEUSEAU BASIC INSPECTOR

Basic programming is sometimes more exasperating than it should be. Programs race to their untimely demise and all you see is **?NF** or **?RG** or some other not to helpful error message. Sure, you're given the line number, but if there's more than one statement on the line it may be difficult to find the culprit.

Cleuseau helps you attack the BASIC programming problem in a reasonable and strategic manner. A program written with the aid of Cleuseau does not need Cleuseau installed in order to run.

You can set stop points anywhere in the program. It's like inserting a **STOP** command, except Cleuseau won't clear the variables. Also, you can set print points anywhere. When a print point is encountered, the line number and the **GOSUB/RETURN** stack is printed. This way you can see where your program was before it encountered an error. The print log can be directed to the printer, to a file or to any other output device.

Cleuseau also has several very powerful BASIC editing commands. It can delete, renumber, move or copy lines, and can also delete **REM** statements and comments, remove all the unnecessary blanks and semi-colons and pack lines together. These commands allow you to keep your BASIC program organized and easy to read. When the time comes you may make your program small and fast, you can delete the unnecessary text and compact your program easily.

Command Descriptions

These Cleuseau commands are used to manipulate, analyze and debug a BASIC program. All commands (except **CTRL_U**, **CTRL_X** and **ESC**) begin with a . (dot). They can be entered only from the BASIC immediate command (direct) mode (Model 100 User's Manual page 99, Tandy 200 BASIC Reference Guide, NEC 8201A N82-BASIC Reference Manual page 1-4). Portions of some commands are optional, depending on the use of the command, and are designated by [ ] square brackets.

Some commands require confirmation. For this, the computer will ask, "**Sure?**". Only a response of "**Y**" or "**y**" will be recognized. Any other key will abort the command.

Most commands operate on a range of lines. Parts of the range or the complete range specification may be omitted. The range will then use default values.

<range>=[<num a>][-[<num b>]]
If **num a** is omitted, 0 is used.
If **num b** is omitted, the last line number in the program is used.
<range>= a .....only line **a** is used.
<range>= a- .....line **a** to the end is used.
<range>= b .....only line **b** is used.
<range>= -b .....beginning to line **b** is used.
<range>= a-b ....line **a** to line **b** is used.
This is the same syntax as BASIC (M100 pg 152, Tandy 200 pg 40, NEC pg 4-77).

Some commands allow an optional match string and only operate on lines in the **<range>** that contain the specified **<string>**.

<range>[=<string>]
If **[=<string>]** is omitted, it operates on all lines in the range. Two forms of **<string>** can be used. **'<string>** will match string and **REM** text while **<string>** (no quote) will match BASIC tokens and variable names.

CTRL X.....**HELP.** List all of the BASIC INSPECTOR Commands. (Use **CTRL_U** to delete all pending command input.)

ESC.....Entering **ESC** automatically enters the next line number during programming. The next line number is the current line number plus the **STEP** value, set by **.STEP <num>**.

CTRL U.....Delete the current input line.

Edit Commands:

These commands clear the Basic variables, just like editing a line would. **.RENUM**, **.MOVE**, **.COPY**, **.MIN**, **.DELCOM**, **.SQUASH**, **.PACK**, and **.EXPAND**, all update **GOTO**s, **GOSUB**s and other line referencing statements to match the new line numbers. **.KILL** does not change any line numbers. If a KILLed line is later referenced, an undefined line error will occur.

The **.RENUM**, **.MOVE** and **.COPY** commands print the message "MAXLINE=n" where n is the largest new line number allowed by the operation. The command will not be done if any new line number exceeds this maximum.

.RENUM <range> [TO <num>] [STEP <num>]...Renumber the range. All references to the renumbered lines are also renumbered. If **.RENUM** is directed to create numbering that overlaps with current numbering, an error message is printed.

New line numbers start at the **TO <num>**. Increment line numbers by the **STEP <num>**. If **TO <num>** is omitted, then the starting line defaults to the first line in the range specification. If **STEP <num>** is omitted then the current auto line **STEP** value is used (See **.STEP**).

A line starting with a **REM** retains its line number; subsequent lines renumber from it. (Lines starting with a ' style **REM** are renumbered.) Use **.MOVE** if the **REM** lines are to be renumbered too.

If **STEP <num>** is zero, each line is offset by the difference between the **TO <num>** and the first line number in the range specification.

.RENUM will not rearrange the line order. Use **.MOVE** to do that.

Example: **.RENUM 100- TO 1000**
Renumber the lines that are now numbered 100 and higher. Start the renumbering with the number 1000. Increment the line numbers by the current auto line **STEP** value.

Example: **.RENUM**
Renumber the entire program. New line numbers start at zero and increment by the current auto line **STEP** value.

.MOVE <range> [TO <num>] [STEP <num>]...Move a range of lines. This command will renumber the lines as they are moved. If lines are moved to an area that already has lines, an error message is printed. Groups of lines can be resequenced with this command. **REM** lines are always renumbered. See **.RENUM** for **TO <num>** and **STEP <num>** defaults.

Example: **.MOVE 30-450 TO 10000 STEP 10**
Move lines 30 through 450 to 10000 and up. Lines 451 through 9999 remain where they are. This is useful for making code into a subroutine.

Example: **.MOVE 1000-1999 TO 10000 STEP 0**

Shift lines 1000 through 1999 to 10000 and up by adding 9000 to each line number.

Example:
```
10 I=0
20 I=I+1:GOSUB 200: GOSUB 100
30 IF I<9 THEN 20
40 END
100 PRINT I
110 RETURN
200 PRINT "I=";
210 RETURN
```

.MOVE 100-110 TO 300

```
10 I=0
20 I=I+1:GOSUB 200: GOSUB 300
30 IF I<9 THEN 20
40 END
200 PRINT "I=";
210 RETURN
300 PRINT I
310 RETURN
```

.COPY <range> [TO <num>] [STEP <num>]...Copy a range of lines and put the copy in another section of the program. GOTOs, GOSUBs and line number arguments INSIDE the copied lines are changed IF THEY REFERENCE LINES IN THE RANGE COPIED.

If lines are copied to an area already occupied by lines, an error message is printed. Attempting to copy a range of lines to a line within the copied range also produces an error. See .RENUM for TO <num> and STEP <num> defaults.

Example:
```
10 I=1
20 IF I=4 THEN 60
30 I=I+1
40 GOSUB 1000
50 GOTO 20
60 END
1000 PRINT I
1010 RETURN
```

.COPY -50 TO 100
(Assumes default step is 10)

```
10 I=1
20 IF I=4 THEN 60
30 I=I+1
40 GOSUB 1000
50 GOTO 20
60 END
100 I=1
110 IF I=4 THEN 60
120 I=I+1
130 GOSUB 1000
140 GOTO 110
1000 PRINT I
1010 RETURN
```

When your program is complete, use **.MIN** to compress it to its smallest volume. It will then use a MINimal amount of memory, saving those precious bytes for other things.

NOTE: Make sure you are finished with the program, or keep an uncompressed backup of it for future modifications. It will be very hard to read or change, after it is compressed.

If you require extra space during programming, use either **.DELCOM**, **.SQUASH**, or **.PACK** on a range of lines to recapture "lost" space.

.MIN [<num>]....Compress a program to its smallest size. Command requires confirmation. See **.PACK** for definition of **<num>**. This command is the same as the command sequence:

```
.DELCOM
.SQUASH
.PACK [LEN <num>]
.RENUM TO 1 STEP 1
```

Note: If the program has been previously packed it may pack better if it is EXPANDed before MINimizing it.

.DELCOM <range>...Delete all comments in the range. Command requires confirmation. You may want to save a backup of your program containing comments.

A line starting with **REM** or ' (quote) is deleted. **GOTOs**, **GOSUBs** and other line referencing statements to this line are redirected to the following line. A **REM** or ' (quote) NOT at the beginning of a line is removed and the line is retained. The last line in the range is NEVER deleted. Comments on **GOSUBs** and **GOTOs** are also deleted.

Example:
```
10 REM sample prog
20 GOSUB 100 DO IT
30 END:REM done
100 'subroutine-do it
110 RETURN 'That's it

.DELCOM
Sure? Y

20 GOSUB 110
30 END
110 RETURN
```

.SQUASH <range>...Remove blanks, unnecessary end quotes on string constants (M100 pg 104, T200 pg 4, NEC pg 3-12) and unnecessary semi-colons in print statements. Command requires confirmation. You may want to save a backup of your program before using **.SQUASH**. Contents of string constants, **REM** statements and **DATA** statements are not changed.

Example:
```
10 PRINT "Hi there."
20 FOR I=1 TO 4:PRINT "I=";I:NEXT
30        END

.SQUASH
Sure? Y

10 PRINT"Hi there.
20 FORI=1TO4:PRINT"I="I:NEXT
30 END
```

.PACK <range> [LEN <num>]...Pack as much on each program line as is possible. Saves 3 or 4 bytes for each line packed. Command requires confirmation. The **LEN <num>** sets the maximum amount to pack on each program line. If omitted, 255 is used. The line numbers of all lines initially longer than **<num>** are listed. (The minimum pack LENgth is 10 and the maximum is 255.) The length of the line's number is always five. This means that lines with shorter line numbers may not pack fully.

The line following a line containing a **REM**, '(quote) or **IF** statement will not be packed onto the end of the previous line. TARGET LINES of **GOTOs**, **GOSUBs** and other line referencing statements are not packed onto the previous line.

Example:
```
10 CLEAR 'program
20 CLS
30 X=1
40 X=X*2
50 PRINT "Hello
60 PRINT X
70 IF X<100 THEN 40
80 END

.PACK
Sure? Y
```

(continued next page)

```
10 CLEAR 'program
20 CLS:X=1
40 X=X*2:PRINT "Hello":PRINT X:IF X<100 THEN 40
80 END
```

Note: If the program has been previously packed it may pack better if it is EXPANDed before PACKing it.

.EXPAND <range>...Unpack multiple statement lines onto multiple program lines, if possible. Command requires confirmation. It may be necessary to RENUMber before EXPANDing in order to fully unpack a program. For example, if you try to EXPAND line 12 and there is already a line 13, line 12 has nowhere to EXPAND and remains PACKed.

Example:
```
10 X=0:FOR I=1 TO 10:X=X+I*I:NEXT
13 END

.EXPAND
Sure? Y

10 X=0
11 FOR I=1 TO 10
12 X=X+I*I:NEXT
13 END
```

.KILL <range>...Delete ALL lines in the range. This command requires confirmation. .KILL will NOT renumber references to the killed lines. If killed lines are later referenced to, an error message is printed.

Example:
```
10 REM Sample
20 PRINT "Hi"
30 PRINT "Bye"
40 END

.KILL 20-
Sure? Y

10 REM Sample
(Only one line remains)
```

Debugging Commands:

These commands do NOT clear BASIC variables. They are allowed only in immediate command (direct) mode (M100 pg 99, T200 pg 1, NEC pg 1-4).

Use the following sample program to help you understand how **.STOP**, **.PRINT**, **.CONT**, and **.OFF** operate. (DATA lines are unaffected by these commands.)

```
5 T=0
10 FOR I=1 TO 8
11 READ X
15 T=T+X*I
20 NEXT
30 PRINT T
50 DATA 1,2,3,4,5,6,7,8

.PRINT 15-
.STOP 10
LIST

5 T=0
10 |FOR I=1 TO 8
11 READ X
15 _T=T+X*I
20 _NEXT
30 _PRINT T
50 DATA 1,2,3,4,5,6,7,8

.CONT 20-40
.OFF 15
LIST

5 T=0
10 |FOR I=1 TO 8
11 READ X
15 T=T+X*I
20 ¯NEXT
30 ¯PRINT T
50 DATA 1,2,3,4,5,6,7,8

.STOP -11==
.PRINT=READ
LIST

5 |T=0
10 |FOR I=1 TO 8
11 _READ X
15 T=T+X*I
20 ¯NEXT
30 ¯PRINT T
50 DATA 1,2,3,4,5,6,7,8
```

In the following debugging commands, **<range>** may be specified. If **<range>** is not specified, the entire range is affected. If **=<string>** is specified, only lines in the range containing the **<string>** will be affected.

When a program contains print, stop or continue points it can not be run from the main menu. You must use the BASIC RUN command. Attempting to run the program from the main menu will cause a ?SN error when the first "point" is reached.

The .STOP, .PRINT and .CONT commands will not insert a point if the line would exceed 255 chars in length. The line number will be printed whenever a "point" is not inserted for this reason.

.STOP <range>[=<string>]...Insert stop points. These cause the program to stop, as if a **STOP** statement were encountered, and then print the line. The program can be resumed with the **CONT** command.

Stop points will replace print points and continue points. Only one stop, print or continue point is allowed at the beginning of any line. Remove stop points by replacing them with print points or continue points, or by using **.OFF**. Stop points show up as a vertical bar |.

.STOP=X= ...Stops before every assignment of X.

.PRINT <range>[=<string>]...Insert print points. When encountered, these print the line number and the **GOSUB/RETURN** stack. Data is printed on the screen if the **.LOG** file number is zero, or the file isn't open for input. Data is printed on the printer if **.LLOG** was called.

.LOG Print log to the screen.
.LOG <num> Print log to the file number **<num>**.
.LLOG Print log to the printer.

Print points replace stop points and continue points. Only one stop, print or continue point is allowed on any line. Remove print points by replacing them with stop points or continue points, or use **.OFF**. Print points show up as an underbar _.

.CONT <range>[=<string>]...Insert continue points. These do not alter program execution. Continue points are "invisible" points. These points show up as tildes ~ (SHIFT GRPH [) on the Model 100/200 and as backslashes \ on the NEC 8201A. A continue point can be quickly changed to a stop or print point. It does not affect the program, it just speeds up the changing of stop and print points.

Continue points replace stop points and print points. Only one stop, print or continue point is allowed on any line. They should not be removed until debugging is completed. Then they may be deleted using **.OFF**.

If the program being debugged uses the **VARPTR** function, use **.CONT** for the full range before starting the program. This way the BASIC variable area will not shift when changing stop or print points.

.OFF <range>[=<string>]....Delete stop points, print points or continue points.

.LOG.....Send the print point log to the screen.

.LOG [<num>]....Set the print point log file number. If this file is open for output (M100 pg 165, T200 pg 47, NEC pg 4-98), the print point log data will be sent to this file. Otherwise, the data will be sent to the screen. If **<num>** is omitted or is zero, the print points send data to the screen. **<num>** is reset to zero each time BASIC is exited.

.LLOG.....Send the print point log to the printer.

Miscellaneous Commands:

These commands do NOT clear BASIC variables. They are allowed only in BASIC command mode.

.FILES.....On the SCREEN, list all the files with their sizes. The first entry in the .BA section has no name and shows the unsaved BASIC program's size. The first entry in the .DO section also has no name and shows the paste buffer's size.

.LFILES.....On the PRINTER, list all the files with their sizes. (See .FILES.)

.LIST <range>[=<string>]..,On the SCREEN, list all lines in the range that contain <string>. If <string> is omitted, then all lines in the range are printed. To find lowercase text, use .LIST <range>='<string> (quote string). Packed lines are printed with one statement per line for easy reading. IF-THEN-ELSE statements are indented appropriately.

.LLIST <range>=[<string>]...On the PRINTER. (See .LIST.)

.STEP <num>.....Set the auto line number step value. Defaults to 10 initially when Cleuseau is installed in the computer. It resets to 10 when Cleuseau is removed from the computer or there is a cold restart.

. (dot).....Print the next BASIC line to be executed and show the GOSUB/RETURN stack.

CLEUSEAU BASIC INSPECTOR

EDITING

.RENUM <range> [TO <num>][STEP <num>]	Renumber range of lines
.MOVE <range>[TO <num>][STEP <num>]	Move the range of lines
.COPY <range>[TO <num>][STEP <num>]	Copy range of lines
.MIN [<num>]	Minimize program size (<num> defaults to 255 chars/line)
.DELCOM <range>	Delete comments in range
.SQUASH <range>	Unnecessary blanks, end quotes and semi-colons removed
.PACK <range>[LEN <num>]	Pack program (LEN <num> defaults to 255 chars/line)
.EXPAND <range>	Unpack program lines
.KILL <range>	Delete lines in range. (Doesn't update line nos.)

DEBUGGING

.STOP <range>[= <string>]	Insert stop points
.PRINT <range>[= <string>]	Insert print points
.CONT <range>[= <string>]	Insert continue points
.OFF <range>[=<string>]	Delete stop, print or continue points
.LOG [<num>]	Print point log file number
.LOG	Print point log to screen
.LLOG	Print point log to printer

MISCELLANEOUS

CTRL U	Delete input line
CTRL X	HELP. BASIC Inspector
ESC	Next line number
.FILES	Files & sizes (screen)
.LFILES	Files & sizes (printer)
.LIST <range>[= <string>]	Lines in range with <string> (screen)
.LLIST <range>[= <string>]	Lines in range with <string> (printer)
.STEP <num>	Set auto line num step
. (dot)	List next line to execute and GOSUB/RETURN stack

APPENDIX

How to Debug with Cleuseau

Making a program work can be a frustrating task. There are several situations where BASIC provides little or no help. For example:

1. If an error occurs in a subroutine, BASIC does not tell you where the subroutine was called from. To rectify this you must put **PRINT** statements all over your program to see what happened before the error.

2. If an error occurs in one of several statements packed onto one line, BASIC does not tell which "sub-line" caused the error. To figure this out you have to break the packed line into individual lines and rerun your program.

3. Sometimes a variable unexpectedly changes its value. Nothing short of searching the entire program with EDIT will give you an idea as to what is wrong.

WITHOUT Cleuseau, debugging can be very time consuming and unrewarding.

Cleuseau's **.PRINT** command helps you understand where your program has been. When your program encounters a print point, Cleuseau shows the current line number and the **GOSUB/RETURN** stack, tracing your program's path. Program lines containing more than one statement will also show the sub-line position. To add flexibility to the **.PRINT** command, **.LOG** and **.LLOG** control where the printed information goes.

When a packed line contains an error, Cleuseau's . (dot) command shows the current line number and the current sub-line where the error occured.

When a variable is changing unexpectedly, you can use **.LIST=var=** to see all the lines where **var** is assigned a new value. If that isn't sufficient, you can use **.STOP=var=** and run the program. Before assigning **var**, Cleuseau will stop your program and let you investigate. You could use BASIC **STOP** statements at each var assignment, but the act of inserting them clears all BASIC variables, requiring you to restart the program. Cleuseau stop points can be inserted or deleted at ANY time without losing BASIC variables.

Print points and stop points serve complimentary purposes. Print points provide confirmation of program flow without disrupting the program. Stop points halt the program so that you can analyze it more extensively. When stopped (either after encountering a stop point or pressing the **BREAK/STOP** key) you may set more stop points, print points or continue points. This lets you adjust your debugging strategy as you go.

CLEUSEAU BASIC INSPECTOR

EDITING

.RENUM <range> [TO <num>][STEP <num>]	Renumber range of lines
.MOVE <range>[TO <num>][STEP <num>]	Move the range of lines
.COPY <range>[TO <num>][STEP <num>]	Copy range of lines
.MIN [<num>]	Minimize program size (<num> defaults to 255 chars/line)
.DELCOM <range>	Delete comments in range
.SQUASH <range>	Unnecessary blanks, end quotes and semi-colons removed
.PACK <range>[LEN <num>]	Pack program (LEN <num> defaults to 255 chars/line)
.EXPAND <range>	Unpack program lines
.KILL <range>	Delete lines in range. (Doesn't update line nos.)

DEBUGGING

.STOP <range>[= <string>]	Insert stop points
.PRINT <range>[= <string>]	Insert print points
.CONT <range>[= <string>]	Insert continue points
.OFF <range>[=<string>]	Delete stop, print or continue points
.LOG [<num>]	Print point log file number
.LOG	Print point log to screen
.LLOG	Print point log to printer

MISCELLANEOUS

CTRL U	Delete input line
CTRL X	HELP. BASIC Inspector
ESC	Next line number
.FILES	Files & sizes (screen)
.LFILES	Files & sizes (printer)
.LIST <range>[= <string>]	Lines in range with <string> (screen)
.LLIST <range>[= <string>]	Lines in range with <string> (printer)
.STEP <num>	Set auto line num step
. (dot)	List next line to execute and GOSUB/RETURN stack

APPENDIX

How to Debug with Cleuseau

Making a program work can be a frustrating task. There are several situations where BASIC provides little or no help. For example:

1. If an error occurs in a subroutine, BASIC does not tell you where the subroutine was called from. To rectify this you must put **PRINT** statements all over your program to see what happened before the error.

2. If an error occurs in one of several statements packed onto one line, BASIC does not tell which "sub-line" caused the error. To figure this out you have to break the packed line into individual lines and rerun your program.

3. Sometimes a variable unexpectedly changes its value. Nothing short of searching the entire program with EDIT will give you an idea as to what is wrong.

WITHOUT Cleuseau, debugging can be very time consuming and unrewarding.

Cleuseau's **.PRINT** command helps you understand where your program has been. When your program encounters a print point, Cleuseau shows the current line number and the **GOSUB/RETURN** stack, tracing your program's path. Program lines containing more than one statement will also show the sub-line position. To add flexibility to the **.PRINT** command, **.LOG** and **.LLOG** control where the printed information goes.

When a packed line contains an error, Cleuseau's . (dot) command shows the current line number and the current sub-line where the error occured.

When a variable is changing unexpectedly, you can use **.LIST=var=** to see all the lines where **var** is assigned a new value. If that isn't sufficient, you can use **.STOP=var=** and run the program. Before assigning **var**, Cleuseau will stop your program and let you investigate. You could use BASIC **STOP** statements at each var assignment, but the act of inserting them clears all BASIC variables, requiring you to restart the program. Cleuseau stop points can be inserted or deleted at ANY time without losing BASIC variables.

Print points and stop points serve complimentary purposes. Print points provide confirmation of program flow without disrupting the program. Stop points halt the program so that you can analyze it more extensively. When stopped (either after encountering a stop point or pressing the **BREAK/STOP** key) you may set more stop points, print points or continue points. This lets you adjust your debugging strategy as you go.

Debugging Strategy

1) Get Started

Set continue points on every line of your program. This is a good way to get started debugging with Cleuseau. To do this, type **.CONT.** Once you have inserted these, you can quickly change them to stop or print points where needed.

2) Establish Program Flow

It is very important to understand what you want your program to do, what you think it will do and what it really does. The three may not be the same. To watch program flow, you could set print points on the entire program and run it. That would create a lot of output and may be useful in cases of extreme need. But, setting print points on lines with **GOSUB**s might tell as much, with less output. Type **.PRINT=GOSUB** to do that. When subroutines are called you'll know where and in what order. To further refine this scheme, use **.PRINT=RETURN.** Now when you run the program, each subroutine will announce its start and end.

3) Capture Bugs

When there is a problem, you need to isolate it. Try to form a hypothesis as to what is causing the failure. Then use stop points when you have an idea of where the problem may be. Set a stop point at the start of the suspect code by typing **.STOP <line number>.** Run the program. When it stops at the stop point, check to see if everything is correct. If it is, immediately set a stop point farther into the suspect code and continue the program by typing **CONT.** Eventually you can locate the problem between two stop points. Zero in on the problem by running the program again and stopping more frequently between the two stop points that isolate the problem.

Sometimes things are already screwed up by the time you reach the first stop point. Try to understand why your hypothesis is wrong. Use information from the program variables to help reformulate that hypothesis. Then set an earlier stop point and run your program again.

4) THINK, THINK, AND THINK

Don't jump to conclusions. Take your time. Cleuseau is a good debugging tool, but YOU must still do the thinking and decision making. When things get very muddled, don't hesitate to change all the stop points and print points back to continue points and start over. A well thought out attack on a program error can yield good, quick results. A sloppy attack will give questionable results and confuse the situation. Take time to reconfirm that "good" code is working as it should. Failing to check "working" code is a common stumbling block. Almost nothing is beyond suspicion.

Useful Debugging Commands

```
.CONT =|.....Change all stop points to continue points.

.CONT =_.....Change all print points to continue points.

.STOP =~.....(Model 100/200)
.STOP =\.....(NEC 8201A)
             Change all continue points to stop points.

.STOP 10000.....Stop before executing line 10000.

.STOP =OPEN.....Stop before opening any files.

.STOP =KILL.....Stop before killing any files.

.STOP =I=.....Stop before any assignment to variable I.

.STOP =POKE.....Stop before poking memory.

.PRINT =~.....(Model 100/200)
.PRINT =\.....(NEC 8201A)
             Change all continue points to print points.

.PRINT.....Get a complete log of program execution.

.PRINT -999.....Log program execution for lines 0 to 999.

.PRINT =CALL....(Model 100/200)
.PRINT =EXEC....(NEC 8201A)
             Log all lines that call machine code routines.

.PRINT =FOR
.PRINT =NEXT.....Log all lines that start or end a FOR loop.

.PRINT =GOSUB 10000....Log all lines that GOSUB 10000.
```

INDEX

for
the Model 100™
or
the TANDY 200™
or
the NEC 8201A™

User's Manual for ROM2 v5.1

Table of Contents — Page

Supplied by: Polar Engineering and Consulting
P. O. Box 7188
Nikiski, Alaska 99635
(907) 776-5529
CIS: 72136,1443

September 1, 1985
(fourth printing)

How to use this Manual

Polar Engineering and Consulting sells ROM2 for the Model 100, the TANDY 200 and the NEC 8201A. Though the three versions of ROM2 are very much alike they are machine specific. A Model 100 ROM2 will not work with an NEC 8201A or a TANDY 200. Likewise, an NEC 8201A ROM2 or a TANDY 200 ROM2 will not work with a Model 100. This manual explains how to use ROM2 on all machines. Variations between the Model 100, the TANDY 200 and the NEC 8201A are noted. For example: ROM2 is accessed from BASIC. The Model 100 ROM2 is accessed by entering CALL 911. The TANDY 200 ROM2 is accessed by entering CALL 921,2. The NEC 8201A ROM2 is accessed by entering EXEC 1124. In this manual this is noted as:

CALL 911	(Model 100)
CALL 921,2	(TANDY 200)
EXEC 1124	(NEC 8201A)

Example programs and debugging sessions are shown for the Model 100, the TANDY 200 and the NEC 8201A.

```
call911
CMD>
Ok
call911
CMD> ?
RUN LOAD KILL SAVE FILES FORMAT
DBG ASMLN ASMN ASML ASM
CMD>

Ok
..
 .RENUM .MOVE .COPY .MIN
 .DELCOM .SQUASH .PACK .EXPAND .KILL
 .LLOG .LOG .OFF .CONT .PRINT .STOP
 .FILES .LFILES .LIST .LLIST .STEP
Ok


Ok
..
 .RENUM .MOVE .COPY .MIN
 .DELCOM .SQUASH .PACK .EXPAND .KILL
 .LLOG .LOG .OFF .CONT .PRINT .STOP
 .FILES .LFILES .LIST .LLIST .STEP
Ok
```

Overview

All ROM2 commands are accessed from BASIC by entering CALL 911 (Model 100), CALL 921,2 (TANDY 200) or EXEC 1124 (NEC 8201A). Hit the BREAK key (Model 100/200) or STOP key (NEC 8201A) to stop ROM2 before or during a command. That returns you back to BASIC. Typing the word MENU as a file name or the DBG command sends you to the main MENU. Typing a question mark (?) prints the keywords recognized by the current command.

All ROM2 commands and all assembler statements recognize upper and lower case as being the same. The CH (change) command is the only exception. Its pattern string and replacement string are both case specific.

Typing the word FILES as a file name for a ROM2 command prints the RAM directory (w/ sizes). This can be very convenient when you've forgotten the name of the file you wanted to use. Unfortunately, ROM2 only operates on RAM .DO files. Transfer files from cassette tape or floppy disk to RAM in order to use ROM2 on them.

About Printers (Model 100 only)

ROM2 supports the parellel printer interface. If you are using a serial interface printer, you can get assembler listing by using the following BASIC program. All LCD output is echoed to the RS232 port during the ROM2 ASM command. (Exercise caution when entering this program. If entered incorrectly, it can cause the Model 100 to lock up and require you to do a COLD-RESTART. Save any valuable files before trying this program the first time.)

```
10 CALL 913,129          'init ROM2
20 ST$="57I1E"           'RS232 STAT
30 CF%=1
40 HL%=VARPTR(ST$)
50 HL%=PEEK(HL%+1)+PEEK(HL%+2)*256-65536
60 CALL 64902,0,6118     'set serial STAT
70 AL%=PEEK(64226)
80 AH%=PEEK(64227)
90 POKE 64226,50
100 POKE 64227,110       'echo to RS232
110 CALL 911'asm
120 POKE 64226,AL%
130 POKE 64227,AH%       'end echo
140 END
```

The value of ST$ will depend on your serial printer's baud rate and protocol. A similar program can echo LCD output while debugging. Insert these two lines when using the TRS-80 Model 100 Disk/Video interface.

```
65 SCREEN 0
135 SCREEN 1
```

Hardware Installation (Model 100/TANDY 200)

WARNING: Improper installation may erase RAM memory or damage ROM2 or the Model 100/200. ROM2 is sensitive to static electricity so exercise caution and FOLLOW installation instructions. READ STEPS 1 THROUGH 10 BEFORE STARTING.

Handling: Do not touch ROM2's pins or the metal contacts on the circuit board. DO NOT REMOVE THE CIRCUIT BOARD; it is essential to ROM2's functionality. Repeated installations may damage the circuit board.

Step 1: Place a soft towel on a flat table surface.

Step 2: Turn the Model 100/200 OFF and place it upside down on the towel. Orient the Model 100/200 with the removable plastic panel nearest you.

Step 3: Insert a penny into the slot on the near side of the panel and pry the panel off.

Step 4: Correctly orient ROM2 with its label reading from left to right. **LOOSELY** place ROM2 into the option ROM socket (M11, it is the nearer of the two). DO NOT USE FORCE IN THIS STEP!

Step 5: ROM2's label should read normally. If **NOT** then go back to the previous step and repeat it.

Step 6: Press ROM2 evenly and steadily with your thumbs into the socket until it will not go down any more.

Step 7: Replace the panel.

Step 8: Turn the Model 100/200 face up ready for typing.

Step 9: Turn on the power. If the main menu does not appear, turn off the power immediately and remove ROM2 and repeat entire installation from the first step.

Step 10: If the main menu did appear then the hardware installation is done. Proceed to the software installation.

Hardware Installation (NEC 8201A)

WARNING: Improper installation may erase RAM memory or damage ROM2 or the NEC 8201A. ROM2 is sensitive to static electricity so exercise caution and FOLLOW installation instructions. READ STEPS 1 THROUGH 12 BEFORE STARTING.

Handling: Do not touch ROM2's pins. Three pins on ROM2 have been modified. This is correct and necessary. DO NOT ATTEMPT TO ALTER ROM2 IN ANY WAY!

Step 1: Place a soft towel on a flat table surface.

Step 2: Turn the NEC 8201A and place it upside down on the towel. Orient the NEC 8201A with the removable plastic panel nearest you.

Step 3: Remove the three philips screws and the plastic cover.

Step 4: Correctly orient ROM2 with its label reading from left to right. Rotate it 90 degrees counter-clockwise. Place it on the second socket from the left. (The first socket has the MicroSoft ROM in it.)

Step 5: ROM2's label should read the same way as the MicroSoft ROM. If **NOT** then go back to the previous step and repeat it.

Step 6: Using both thumbs, press ROM2 firmly into its socket. Make sure that all the pins are inserted.

Step 7: Replace the panel.

Step 8: Turn the NEC 8201A face up ready for typing.

Step 9: Turn on the power. If the main menu does not appear, turn off the power immediately and remove ROM2 and repeat entire installation from the first step.

Step 10: If the main menu did appear then the hardware installation is done. Proceed to the software installation.

Software Installation

The ROM2 software is accessed from BASIC. First, test ROM2 by entering:

CALL 913,193	(Model 100)
CALL 921,194	(TANDY 200)
EXEC 1124,193	(NEC 8201A)

If no number appears within five seconds then your ROM2 may not be installed properly. Repeat the hardware installation instructions. If the number printed does not match the five digit number on ROM2's label then it may not be installed properly. (It may be necessary to COLD-RESTART the Model 100/200 or NEC 8201A before reinstalling ROM2.)

Type the following BASIC command line to access ROM2.

CALL 911	(Model 100)
CALL 921,2	(TANDY 200)
EXEC 1124	(NEC 8201A)

ROM2 then prompts with "CMD>". Now you can select one of these possible commands.

FILES	files directory (with file sizes)
COPY	copy .DO file contents
FEQ	compare two .DO files
CH	global string change for a .DO file
RN	renumber the current basic program (Model 100/200 only)
ASM	macro assembler (LCD listing)
ASML	macro assembler (LPT listing)
ASMN	macro assembler (LCD, list errors only)
ASMLN	macro assembler (LPT, list errors only)
DBG	symbolic debugger
?	list the names of the commands

Each of these commands is defined in detail in the following sections.

Since CALL 911 (Model 100), CALL 921,2 (TANDY 200) or EXEC 1124 (NEC 8201A) must be done before selecting any ROM2 command it will be convenient to define function key 6. Use the BASIC KEY command to do that.

KEY 6,"CALL911"+CHR$(13)	(Model 100)
KEY 6,"CALL921,2"+CHR$(13)	(TANDY 200)
KEY 6,"EXEC1124"+CHR$(13)	(NEC 8201A)

Press function key 6 and then function key 1 (Model 100/200) or function key 3 (NEC 8201A) and you will see the RAM directory complete with sizes.

From BASIC programs ROM2 commands can be accessed like so:

Model 100:	TANDY 200:	NEC 8201A:
CALL911'FILES	CALL921,2'FILES	EXEC1124'FILES
CALL911'COPY	CALL921,2'COPY	EXEC1124'COPY
or		
ST$="FILES" + CHR$(13)	ST$="FILES" + CHR$(13)	
CALL913,17,VARPTR(ST$)	CALL921,18,VARPTR(ST$)	no equivalent

FILES (directory of all RAM files)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> FILES
```

Typical LCD screen output from this command might look like this.

```
.BA:.....345 BACKUP.1617
.DO:.....86 HWINST.1940 SWINST.1852
    COPY...1415 FEQ....1189 RN.....3561
    ASM....3864 ALC....2791
14867 Bytes free
```

The directory is listed in three major groups. First the .BA (BASIC programs), then the .DO (document files), then the .CO (8085 code programs), and finally, the number of free bytes. When there are no files in the .CO group, that group is omitted.

Following each file name is the number of RAM bytes that file is using. This is called the size of the file. In other words, killing a file increases the number of free bytes by its size. Also copying a file decreases the number of free bytes by its size.

The first entry in the .BA: section has no name and shows the unsaved BASIC program's size. The first entry in the .DO: section has no name and shows the paste buffer's size.

COPY (duplicate a .DO file)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> COPY file a file b
```

The purpose of this command is to create a new (or overwrite an old) 'file_b'.DO and duplicate the contents of 'file_a'.DO into it. Suppose that file MEMO1.DO exists and we want a copy of it called MEMO2.DO. That is done by typing:

```
Ok
(function key 6)
CMD> COPY MEMO1 MEMO2
```

This creates a new file called MEMO2.DO which contains a copy of the text that is in MEMO1.DO. When MEMO2.DO already exists, rewrite confirmation is needed. For example:

```
Ok
(function key 6)
CMD> COPY MEMO1 MEMO2
Rewrite? Y
```

overwrites the previous contents of MEMO2.DO with a copy of the contents of MEMO1.DO. In the next sequence MEMO2.DO is saved and not overwritten.

```
Ok
(function key 6)
CMD> COPY MEMO1 MEMO2
Rewrite? N
File: MEMO3
```

The previous command is equivalent to this:

```
Ok
(function key 6)
CMD> COPY MEMO1 MEMO3
```

When there isn't enough free memory to copy the file, the 'Out of memory.' message is printed.

FEQ (compare two .DO files)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> FEQ file a file b
```

This command compares the characters in the 'file_a'.DO with the characters in 'file_b'.DO. Suppose that the file A.DO (which is 54 lines long) has just been copied (using the COPY command or the paste buffer) into both files B.DO and C.DO. Futhermore, the file C.DO has been altered using EDIT such that the 24th character in the first line is now gone. First compare A.DO with B.DO.

```
Ok
(function key 6)
CMD> FEQ A B
54.Yes.
```

The **54** shows how many complete lines matched. The **Yes.** shows that the contents of A.DO are identical to the contents of B.DO. Now compare A.DO with C.DO.

```
Ok
(function key 6)
CMD> FEQ A C
0.23 (beep)
```

In this case the files have no completely matching lines (before they differ) and only match through the first 23 characters. Please note that each end-of-line in a .DO file is stored as two characters (a carriage return, ASCII 13, followed by a line feed, ASCII 10). Thus the character match count shown by FEQ is the sum of characters matched plus two times the number of lines matched.

CH (global string change)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> CH file
From:   pattern string
To:     replacement string
```

The purpose of this command is to find all the occurances of 'pattern_string' in 'file'.DO and change them to 'replacement_string'. The strings in this command are case sensitive, hence a 'pattern_string' must match exactly (lowercase to lower, uppercase to upper) in order for the substitution to occur. Also, leading and trailing blanks are used in both strings. Entering a null 'replacement_string' deletes all occurances of the 'pattern_string'. A single question mark (?) is treated as a 'pattern_string' or a 'replacement_string' and will not list the command options, since there are none.

Suppose that file MEMO1.DO exists and we want a copy of it called MEMO2.DO with the name Joe Smith replaced by Bob Jones, Jr. That is done by typing:

```
Ok
(function key 6)
CMD> COPY MEMO1 MEMO2
Ok
(function key 6)
CMD> CH MEMO2
From:   Joe Smith
To:     Bob Jones, Jr.
```

Now MEMO2.DO has the name Bob Jones, Jr. everywhere the name Joe Smith was.

Control Characters

Control characters can be specified in both the 'pattern_string' and the 'replacement_string'. An ^ (uphat, shift 6) indicates that the following character specifies a control character. For example ^M specifies control-M (carriage return, ASCII 13) while ^J specifies control-J (line-feed, ASCII 10). Two uphats (^^) specify a literal uphat. When the last character in a pattern is a single uphat, it is a literal uphat. Specifying the end-of-file control character, ASCII 26, (^Z is used as the end-of-file marker) is not allowed because manipulating the end-of-file marker is too dangerous.

One line can be split into two by using ^M^J in the 'replacement_string'. Two lines can be concatenated by using ^M^J in the 'pattern_string'. The 'pattern_string' ^Jtext matches text at the start of a line. The 'pattern_string' text^M matches text at the end of a line. Be sure to replace the ^J or ^M character when doing this type of substitution. Many other line manipulating operations can also be done.

RN (renumber a BASIC program--Model 100/200 only)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> RN
F,L,T,S: first,last,to,step
```

With this command all or part of the current BASIC program can be renumbered. The current BASIC program is the last program loaded. Or it is the unsaved BASIC program when no program has been loaded since entering BASIC.

first and last: These two parameters are used to specify the renumbering range. All BASIC lines with line numbers greater than or equal to 'first' and less than or equal to 'last' are renumbered. When 'first' is not specified, 10 is used. When 'last' is not specified, 65535 is used.

to: This parameter specifies the new line number to be assigned to the first line that gets renumbered. When 'to' is not specified, the value of 'first' is used.

step: The absolute value of this parameter specifies the increment or spacing between the lines after renumbering. When 'step' is positive, no REM lines in the range are renumbered (single quote style rem lines are always renumbered). Lines immediately following a REM line are renumbered relative to the REM line number. When 'step' is negative, -'step' (minus 'step') is used as the increment and REM lines in the range are renumbered. When 'step' is not specified, +10 is used.

Suppose that the following program has been entered and was the current BASIC program.

```
10 'Hailstone numbers
15 INPUT"Starting value";N
18 PRINT"N=";N
19 IF N=1 THEN 999
20 IF N MOD 2=0 THEN N=N/2 ELSE N=N*3+1
21 GOTO 18
999 REM Done.
```

Now suppose that we want to count the 'Hailstone' numbers as they are generated. To do so we need to insert two lines and modify a third.

```
        12 I=1
        20.5 I=I+1
modify  18 PRINT "I=";I;" N=";N
```

Unfortunately, BASIC does not allow fractional line numbers. Renumbering remedies this problem.

```
Ok
(function key 6)
CMD> RN
F,L,T,S: _       (just press ENTER)
```

The BASIC program now looks like this.

```
10 'Hailstone numbers
20 INPUT"Starting value";N
30 PRINT"N=";N
40 IF N=1 THEN 999
50 IF N MOD 2=0 THEN N=N/2 ELSE N=N*3+1
60 GOTO 30
999 REM Done.
```

Note that the GOTO statement number in line 60 was changed to the appropriate new value (30 instead of 18). Now it is easy to insert the changes.

Also note that line 999 does not get changed to line 70 as one might expect. This feature allows REM lines to retain their original line numbers even after renumbering. When a step factor of minus ten (instead of the default positive ten) is specified, line 999 changes to line 70. Typically, the first line of every subroutine begins with a REM statement. Thus, it is desirable to keep those line numbers constant as the program is developed. However, if necessary, it is not difficult to renumber some or all of the REM lines by using a negative step factor and an appropriate range.

Here are some assorted F,L,T,S specification examples:

	First	Last	To	Step	renumber REM
F,L,T,S: ,,,-20	10	65535	10	20	YES
F,L,T,S: ,1000	10	1000	10	10	NO
F,L,T,S: 1000	1000	65535	1000	10	NO
F,L,T,S: ,,100,100	10	65535	100	100	NO

Finally, when renumbering by a given range and step would result in a non-ascending line number sequence, the line number that makes the sequence non-ascending is printed. Usually a smaller step value or a more limited range will solve this problem.

ASM, ASML, ASMN, ASMLN (macro assembler)

This command is accessed by typing:

Ok
(function key 6)
CMD> ASM assembly source file

The purpose of an macro assembler is to read a text file and translate it into binary machine codes and data. Assembled machine code is placed directly into the area between HIMEM and MAXRAM unless the 'Output File' option is used. ASM and ASML list each line as it is assembled. ASMN and ASMLN list only lines with errors. ASM and ASMN list to the LCD screen. ASML and ASMLN list to the parallel printer port.

CAUTION: Be sure to save source files and other important RAM data on cassette tape when developing assembly programs. This must be done because during the development phase executing a machine code program containing errors can destroy RAM data or even cause a COLD-RESTART.

Assembly Language Syntax

An assembly language program is a text file composed of single line machine instructions represented in symbolic form. Each line has the following stucture:

label: instruction ;comment text

label: is the optional statement label and, when present, it must begin in column one. The word used must be composed of no more than nine of the characters **$0123456789?@ABC...XYZ_** and it may end with a colon. A statement label names a line for reference elsewhere in the program.

instruction is an optional specification of an 8085 machine instruction or an assembler instruction. This instruction is composed of an 8085 opcode (or pseudo op) and its required number of arguments. Again, lowercase opcode names are allowed and do match their uppercase names.

;comment is also optional and is used for documenting and otherwise clarifying the operational characteristics of the program. It must begin with a semi-colon.

Argument expressions may use the symbol **$**. It represents ASM's current PC value and is most often used in DB statements when calculating offsets from a statement label defined earlier.

Pseudo Ops

DB expr [,expr]*

Assigns one or more bytes with the values of the 'expr's. Only the DB pseudo op may have an 'expr' that is a quoted string of arbitrary length. This allows for easy message definition.

DS expr

Allocates space (defines storage) for a variable, a table or an array. The value of 'expr' is added to PC, reserving that many bytes for the program. Each reserved byte is initialized to the value zero.

DW expr [,expr]*
Assigns one or more two-byte words with the values of the 'expr's.

ELSE
Begins a conditional assembly ELSE block. Code is generated from this statement to its matching END when the corresponding IF is false.

END
Signals the end of an IF, IFZ, IFNZ, ELSE, MAC statement or the program.

label: EQU expr
Assigns (equates) the value of 'expr' to 'label:'. This pseudo op requires a statement label.

IF expr
Begins a conditional assembly IF block. Code is generated from this statement to its matching END (or ELSE) when 'expr' is greater than or equal to zero.

IFZ expr
Begins a conditional assembly IFZ block. Code is generated from this statement to its matching END (or ELSE) when 'expr' is equal to zero.

IFNZ expr
Begins a conditional assembly IFNZ block. Code is generated from this statement to its matching END (or ELSE) when 'expr' is not equal to zero.

MAC
Begins a macro definition block. The contents of this block (terminated by its matching END) are substituted into the assembly by the &label: invocation statement.

ORG expr
Sets the PC to the value of 'expr'. Used for defining the first instruction's address (origin).

&RAM .DO file
Includes file. Inserts the 'RAM_.DO_file' into the assembly at the point of this statement. Does not list the file's lines as it is processed.

&&RAM .DO file
Includes file. Inserts the 'RAM_.DO_file' into the assembly at the point of this statement. Lists the file's lines as it is processed when the include statement itself is printed.

&label: [arg [;arg]*]
Invokes macro. Substitutes the arguments specified into the 'label' macro and insert it into the assembly. Does not list the macro's lines as it is processed.

&&label: [arg [;arg]*]
Invokes macro. Substitutes the arguments specified into the 'label' macro and insert it into the assembly. Lists the macro's lines as it is processed when the invocation statement itself is printed.

EOF
End of file. Ends the program or included file by providing the necessary number of END statements to terminate processing of the file.

Assembler Output Echo

Entering 'ASM file_name /' causes the bytes generated by the assembler to be printed in hex form interlaced with the printed assembly source.

Assembler Output File

Entering 'ASM file_name_1 file_name_2' causes the bytes generated by the assembler to be stored in hex form in 'file_name_2'. All PC checking is turned off and the output is not poked into memory. The output file has the same format as '_ASM_.DO' except for the hex data that is stored by the assembler immediately following the '>' character. (See "ASM/DBG Label Table Format".)

Assembler Include Files

An assembly language program may insert another RAM .DO file. This allows common macros and external labels to be placed in a global include file for sharing amongst many programs. Include has two forms:
1) silent (do not print included file as it is processed)

```
&RAM .DO file
```

or
2) echo (print file if the include statement itself is printed)

```
&&RAM .DO file
```

Include file statements may be nested.

Assembler Macros

A macro definition has the form:

```
label:  MAC
        macro body
        END
```

Invocation of a macro has two forms:
1) silent (do not print macro as it is substituted)

```
&label: arg1;arg2;...argn
```

or
2) echo (print macro if the invocation statement itself is printed)

```
&&label: arg1;arg2;...argn
```

Macro invocation substitutes argn in place of the macro definition's symbolic argument #n found in the macro body. When the macro invocation does not specify an argument, a null string is used. Symbolic argument #0 is replaced by all arguments. Quoted semicolons must not be used within an argument because the quotes are not recognized and the argument will be split. Macros may be invoked from within macros. Since args are substituted, care must be exercised when using args in complex expressions. It may be necessary to parenthesize the symbolic arg to ensure correct expression evaluation. A macro may not in turn define new macros. Invocation of such a macro causes a 'Bad op.' error. Finally, use of a macro label in an expression yields a funny value, because the value of the macro label is the memory address of the macro definition.

Assembler Conditional Code Generation

Conditional assembly is controlled by the IF assembler statement. A conditional assembly block has the form:

```
IF expression
conditional code
END
```

If expression is greater than or equal to zero then the conditional code is assembled. Otherwise it is skipped. Skipped assembly code does not create labels, macros, or machine code. Skipped assembly code need not be valid assembly code. IF blocks may be nested and placed within or around macro definitions. Two other forms of the IF statement are recognized: IFZ and IFNZ. An IFZ block is assembled when expression is zero. An IFNZ block is assembled when expression is not zero. An IF, IFZ, or IFNZ block may terminate with an ELSE assembler statement instead of an END. An IF-ELSE block pair has the form:

```
IF expression
if conditional code
ELSE
else conditional code
END
```

The else conditional code is assembled when the if conditional code is skipped and visa-versa. The ELSE assembler statement should only be used in conjunction with IF statements. Other uses produce strange results.

IF, IFZ, and IFNZ all allow multiple expressions. The effect is the same as nesting the IFs with one expression per IF in the order specified.

Assembler END Statement

The END statement is used in three contexts: 1) end of macro definition block, 2) end of IF block, and 3) end of program. The end of file provides the required number of END statements in order to terminate processing of that file. The first END statement that is unmatched with a preceding MAC, IF, IFZ, IFNZ, or ELSE statement ends the program. During assembly, when code is being skipped, a '*' instead of a ':' is printed at the beginning of each line. Also, the current nesting level is printed.

Assembler ORG Statement

The ORG statement sets the assembler's PC. For programs that have only one ORG statement and execute starting at that statement, there is a general purpose method for creating .CO files:

```
SAVEM"file",PEEK(64872)+PEEK(64873)*256,          (Model 100)
            PEEK(64870)+PEEK(64871)*256-1,
            PEEK(64872)+PEEK(64873)*256

SAVEM"file",PEEK(63592)+PEEK(63593)*256,          (TANDY 200)
            PEEK(63590)+PEEK(63591)*256-1,
            PEEK(63592)+PEEK(63593)*256

BSAVE"file",PEEK(64616)+PEEK(64617)*256,          (NEC 8201A)
            PEEK(64614)+PEEK(64615)*256-
            (PEEK(64616)+PEEK(64617)*256),
            PEEK(64616)+PEEK(64617)*256
```

Macro Assembler Source

The file read by the macro assembler contains symbolic descriptions of both 8085 instructions (opcodes with arguments) as well as macro assember instructions (called pseudo ops). If you do not have complete 8085 instruction documentation or you are a beginning assembly language programmer it is suggested that you buy "8080A/8085 Assembly Language Programming", by Lance A. Leventhal. It is available in the computer science section of most large book stores. Also, it can be ordered by mail from Osborne/McGraw-Hill, 2600 Tenth Street, Berkeley, CA 94710 as ORDER #10-1 for $18.95. Shipping is $1.50 UPS, $3.00 1st class/UPS Blue Label. California residents add local sales tax. The opcodes and pseudo ops provided by the ROM2's macro assembler conform to the ones used in Leventhal's book and are recognized as the 8080/8085 standards.

DBG (symbolic debugger)

This command is accessed by typing:

```
Ok
(function key 6)
CMD> DBG
Polar Engr & Cons (C)1985
DBG> DBG command
```

CAUTION: Be sure to save source files and other important RAM data on cassette tape when developing assembly programs. This must be done because during the development phase executing a machine code program containing errors can destroy RAM data or even cause a COLD-RESTART.

DBG commands:

Simulate:	GE, GO, IN, N, OUT, S, SE
Execute:	RUN
PC control:	CE, SKP
Break point:	CB, CBS, LBS, 1, 2, 3, 4
Memory inspection:	EX, L, MAP
Register status:	D
Output control:	LCD, LPT
Trace:	TN, TY
Main Menu:	MENU
Command list:	?
Assembly:	see immediate & patch

DBG starts simulating on code that 'gracefully' returns to BASIC. Entering GE, GO, OUT, and RUN all send you back to BASIC.

Simulation vs. Execution

There are two things to consider when debugging machine code. The first is speed of simulation. The time required to simulate a single instruction is about 1000 times that of executing the instruction. This is an obvious disadvantage. If a subroutine has already been debugged then it is not necessary to spend an excessive amount of time simulating it. However, no break points are recognized while executing code, so simulation does serve a very useful purpose.

The second problem with simulation is that some code just cannot be properly simulated. Take for example a subroutine that displays a character on the LCD screen. Here is a conflict in the use of a shared resource, the LCD screen. When such a subroutine is stepped through, the situation on the screen changes and therefore the simulation does not have the same result as the execution of the same code. Be aware that for this reason some simulations will not work.

DBG Simulate Commands

GE [expr]: Simulates all instructions until 'expr' break points have been encountered.

GO [expr]: Simulates all but CALL instructions until 'expr' break points have been encountered. (Subroutines are executed, not simulated.)

IN [expr]: Simulates all instructions until 'expr' CALL and/or RET instructions have been simulated. Simulation stops when any break point is encountered.

N [expr]: Simulates all instructions until 'expr' instructions have been simulated. Simulation stops when any break point is encountered.

OUT [expr]: Simulates all but CALL instructions until 'expr' RET instructions have been simulated. (Subroutines are executed, not simulated.) Simulation stops when any break point is encountered.

S [expr]: Simulates all but CALL instructions until 'expr' instructions have been simulated. (Subroutines are executed, not simulated, and counted as one instruction.) Simulation stops when any break point is encountered.

SE [expr]: Simulates all instructions until 'expr' instructions have been simulated. (Subroutines are simulated but counted as only one instruction.) Simulation stops when any break point is encountered.

BREAK or **STOP**: Hitting the BREAK key (Model 100) or STOP key (NEC 8201A) interrupts simulation. (PAUSE key or CTRL-S suspends simulation.) The BREAK key or STOP key can not interrupt execution.

The count expression is optional for these seven simulation commands. When simulation is interrupted by hitting the BREAK key and the remaining count is more than one, the bell sounds, the remaining number is printed, and DBG prompts for a new command. If the next simulate command is entered without a count expression, the number printed when the BREAK key was hit is used. The simulate command count is one when DBG is first entered.

The simulate commands GO, OUT, and S all execute subroutines when a CALL instruction (or any flag testing variant) is encountered. (Standard ROM RST 0,2,3,4,5,6 and Option ROM RST 0,1,4,6,7 instructions are considered calls.) The subroutine invoked is not simulated. Instead, it is executed directly by the 8085 processor in the Model 100. After the subroutine is done, DBG simulates the instruction that follows the CALL.

Before a subroutine is executed, DBG overwrites the called subroutine's return with a trap return and saves the subroutine's return elsewhere. When the subroutine returns to the trap return, the subroutine's real return is used as the next PC for simulation.
Note: A subroutine that does memory referencing through the stacked return address will not work because during the execution of the subroutine, DBG's trap return is on the stack instead of the subroutine's return. Also, a subroutine that returns more than one level exits DBG (like a RUN command), since DBG's trap return is skipped. Use simulate commands GE, IN, N, and SE to avoid these two cases.

Note: See "Appendix J: DBG Simulation Examples" for additional help.

DBG Non-Simulate Commands

CB: Clears the current break point. If a break point was just encountered, clear it (set it to zero).

CBS: Clears all four break point addresses. (Sets them to zero.)

CE [expr]: Simulates entering a subroutine with a CALL instruction by stacking the contents of PC and placing the value of 'expr' in PC. Displays new register status.

D: Displays the current register status. It redisplays the current register status as if simulation has just stopped. (See "DBG Register Status".)

EX [expr]: Examines memory. Shows the two byte word's value whose address is 'expr'. (Also shows the byte value.)

Format: xxxx/ddddd=xx xxxx aa
1 2 3 4 56

Shown are: 1) hexadecimal value of 'expr', 2) decimal value of 'expr', 3) hexadecimal value of byte at 'expr' memory location, 4) hexadecimal value of word at 'expr', 5) ASCII byte at 'expr', and 6) ASCII byte at 'expr'+1. (Note: This command is also a handy hexadecimal-decimal calculator.)

L [expr]: Disassembles memory. It lists instructions starting at address 'expr'. Hit the BREAK key to terminate list.

Format: xxxx 8085 instruction
xxxx 8085 instruction
...
1 2

Shown are: 1) hexadecimal address, and 2) 8085 instruction at that address. (Note: All values within the 8085 instruction are shown in hexadecimal.)

LBS: Lists the four break point addresses.

Format: 1.xxxx 2.xxxx 3.xxxx 4.xxxx

Shown are the four addresses in hexadecimal.

LCD: Sends all status information to the LCD screen. (not LPT)

LPT: Sends all status information to the printer. (not LCD)

MAP [expr]: Displays the memory map starting at 'expr'. This command prints a hexadecimal and ASCII memory dump. Hit the BREAK key to terminate MAP.

Format: xxxx xx xx xx xx xx xx xx xx xx aaaaaaaa
xxxx xx xx xx xx xx xx xx xx xx aaaaaaaa
...
1 2 3

Shown are: 1) hexadecimal address, 2) hexadecimal byte values in memory from that address to the that address plus seven, and 3) ASCII values of those same memory locations.

MENU: Exits to the main menu.

RUN: Starts executing instructions. (DBG permanently exited.)

SKP: Advances PC to the instruction that immediately follows the current one. Displays new register status.

TN: Stops tracing simulation. (not TY)

TY: Begins tracing simulation. The register status is printed after every simulated instruction. (not TN)

1 [expr] (2, 3, or 4): Sets the corresponding break point address to the value of 'expr'.

Note: When 'expr' is omitted the current value of PC is used.
Note: a is an ASCII character (control-characters displayed as a dot).

DBG Immediate and Patch Assembly Commands

Immediate assembly commands are of the form:

8085 instruction

When DBG gets a command that does not start with an expression and is not one of its special command keywords, DBG sends the text to the assembler. If the command is a valid '8085_instruction' then DBG simulates it.

The patch command form is:

expr,8085 instruction

or

expr,DB expr,...

or

expr,DS expr

or

expr,DW expr,...

When DBG gets a command that starts with an expression, DBG sends the text to the assembler. If the command is a valid '8085_instruction' or DB, DS, or DW pseudo op then DBG deposits the assembled, code starting at the memory location indicated by 'expr'.

The $ symbol (symbolic assembler PC) is set to one more than the last address patched. If there is a syntax error then it is unchanged. Immediate assembly commands may use the $ symbol in expression arguments. Entering just an expression in DBG sets the $ symbol to that value. (Constants 1, 2, 3, and 4 are interpreted as set break point commands.)

Example patch assembly commands:

```
57500,MVI A,129
$,CALL 913
```

Addresses 57500 and 57501 are patched by the first command. After which the $ symbol is 57502. Thus, the next three bytes in memory are readily patched with the second command.

DBG PC

DBG allows expressions with the symbol PC. Its value is always the next instruction's address. Relative jumps are done by entering 'JMP PC+offset'. Also, 'L PC' (or just 'L') displays the instructions about to be simulated.

DBG Register Status

The current register status is displayed in this format

```
xxxx instruction zf cf pf s SP=xxxx.yyyy
 A=xx BC=xxyy.zz DE=xxyy.zz HL=xxyy.zz
```

where xxxx, yyyy, xx, xxyy and zz are hexadecimal numbers.

xxxx instruction: The 8085 instruction at memory location xxxx is next to be simulated by DBG (xxxx is the current value of the program counter or PC). When ROM2 is enabled (standard ROM disabled), an exclamation point is printed immediately following the current PC.

zf cf pf s: Each of the four current flag values is shown here; zf (zero flag): NZ or Z, cf (carry flag): NC or C, pf (parity flag): PE or PO, and s (sign flag): P or M.

SP=xxxx.yyyy: The stack pointer, SP, contains the word value xxxx and the word at memory location xxxx contains the word value yyyy.

A=xx: The A register contains the byte value xx.

BC=xxyy.zz: The BC register pair contains the word value xxyy, the B register contains the byte value xx, the C register contains the byte value yy and the byte at memory location xxyy contains the byte value zz.

DE=xxyy.zz: Refer to BC (D is to B as E is to C).

HL=xxyy.zz: Refer to BC (H is to B as L is to C).

Example:

```
FD8C CALL FAA5    Z NC PE P SP=F160.1E20
 A=04 BC=00BE.50 DE=0000.C3 HL=0850.62
```

The current DBG PC is FD8C hex. The instruction starting at that address is CALL FAA5 (hex). The Zero flag is set, the Carry flag is clear, the parity is even, and the sign is positive. The stack pointer is F160 hex. The last word pushed onto the stack is 1E20 hex. The A register is four. The BC register pair is BE hex (the B register is zero and the C register is BE hex). The byte at address BE hex is 50 hex. (An LDAX B instruction would set the A register to BE hex.) The DE register pair is zero (both the D and E registers are zero). The byte at address zero is C3 hex. (An LDAX D instruction would set the A register to C3 hex.) The HL register pair is 850 hex (the H register is eight and the L register is 50 hex). The byte at address 850 hex is 62 hex and is value of the M register.

8085 Instruction Table

Arguments:

addr -- 16 bit data address
byte -- 8 bit binary value
dreg -- destination (see reg)
drp -- (destination rp for MVX) one of B, D, or H
label -- 16 bit instruction address
n -- (restart index) one of 0, 1, 2, 3, 4, 5, 6, or 7
port -- 8 bit binary input or output port number
reg -- one of A, B, C, D, E, H, L, or M
(the M register is the byte addressed by the HL pair)
rp -- (register pair) one of B, D, H, or SP
(PUSH and POP use rp PSW instead of SP)
sreg -- source (see reg)
srp -- (source rp for MVX) one of B, D, or H
word -- 16 bit binary value

Registers:

A -- accumulator (low byte of PSW)
B -- auxilary register (high byte of B rp)
C -- auxilary register (low byte of B rp)
D -- auxilary register (high byte of D rp)
E -- auxilary register (low byte of D rp)
H -- auxilary register (high byte of H rp)
L -- auxilary register (low byte of H rp)
T -- temporary register (only used during comparisons)

Register pairs:

B -- (BC) B and C registers, high byte is B, low byte is C
D -- (DE) D and E registers, high byte is D, low byte is E
H -- (HL) H and L registers, high byte is H, low byte is L
PC -- program counter
PSW -- program status word, low byte is A
SP -- stack pointer

Flags: (in high byte of PSW)

Cf -- carry (NC:Cf=0, C:Cf=1) bit 0 of PSW high byte
Pf -- parity (PO:Pf=0, PE:Pf=1) bit 2 of PSW high byte
Sf -- sign (P:Sf=0, M:Sf=1) bit 7 of PSW high byte
Zf -- zero (NZ:Zf=0, Z:Zf=1) bit 6 of PSW high byte
TSf -- true sign, bit 5 of PSW high byte

Flag status:

. -- no change
x -- changed according to the result of the instruction's operation
0 -- reset (clear)
1 -- set

Operators:

+ -- addition
- -- subtraction
not -- 0=>1, 1=>0
and -- 0 and 0=>0, 0 and 1=>0, 1 and 0=>0, 1 and 1=>1
or -- 0 or 0=>0, 0 or 1=>1, 1 or 0=>1, 1 or 1=>1
xor -- 0 xor 0=>0, 0 xor 1=>1, 1 xor 0=>1, 1 xor 1=>0

	instruction	action	Zf	Cf	Pf	Sf
	ACI byte	A<=A+byte+Cf	x	x	x	x
	ADC reg	A<=A+reg+Cf	x	x	x	x
	ADD reg	A<=A+reg	x	x	x	x
	ADI byte	A<=A+byte	x	x	x	x
	ANA reg	A<=A and reg	x	0	x	x
	ANI byte	A<=A and byte	x	0	x	x
	CALL label	SP<=SP-2, w[SP]<=PC+3, PC<=label	.	.	.	.
	CC label	if Cf=1 then CALL label	.	.	.	.
	CM label	if Sf=1 then CALL label	.	.	.	.
	CMA	A<=not A	.	.	.	.
	CMC	Cf<=not Cf	.	x	.	.
	CMP reg	T<=A-reg	x	x	x	x
	CNC label	if Cf=0 then CALL label	.	.	.	.
	CNZ label	if Zf=0 then CALL label	.	.	.	.
	CP label	if Sf=0 then CALL label	.	.	.	.
	CPE label	if Pf=1 then CALL label	.	.	.	.
	CPI byte	T<=A-byte	x	x	x	x
	CPO label	if Pf=0 then CALL label	.	.	.	.
	CZ label	if Zf=1 then CALL label	.	.	.	.
	DAA	decimal add adjust	x	x	x	x
	DAD rp	HL<=HL+rp	.	x	.	.
	DCR reg	reg<=reg-1	x	.	x	x
	DCX rp	rp<=rp-1	.	.	.	.
**	DEHL byte	HL<=HL+byte	.	.	.	.
**	DESP byte	DE<=SP+byte	.	.	.	.
	DI	disable interrupts	.	.	.	.
	EI	enable interrupts	.	.	.	.
**	HLMBC	HL<=HL-BC	x	x	h	x
	HLT	halt 8085 processor	.	.	.	.
	IN port	A<=data from port	.	.	.	.
	INR reg	reg<=reg+1	x	.	x	x
	INX rp	rp<=rp+1	.	.	.	.
	JC label	if Cf=1 then JMP label	.	.	.	.
	JM label	if Sf=1 then JMP label	.	.	.	.
	JMP label	PC<=label	.	.	.	.
	JNC label	if Cf=0 then JMP label	.	.	.	.
	JNZ label	if Zf=0 then JMP label	.	.	.	.
	JP label	if Sf=0 then JMP label	.	.	.	.
	JPE label	if Pf=1 then JMP label	.	.	.	.
	JPO label	if Pf=0 then JMP label	.	.	.	.
**	JTM label	if TSf=1 then JMP label	.	.	.	.
**	JTP label	if TSf=0 then JMP label	.	.	.	.
	JZ label	if Zf=1 then JMP label	.	.	.	.
	LDA addr	A<=b[addr]	.	.	.	.
	LDAX B	A<=b[BC]	.	.	.	.

Note: Instructions marked with ** are undocumented 80C85 opcodes.

	instruction	action	Zf	Cf	Pf	Sf
	LDAX D	A<=b[DE]	.	.	.	.
	LHLD addr	HL<=w[addr]	.	.	.	.
**	LHLI	HL<=w[DE]	.	.	.	.
	LXI rp,word	rp<=word	.	.	.	.
	MOV dreg,sreg	dreg<=sreg	.	.	.	.
	MVI reg,byte	reg<=byte	.	.	.	.
	MVX drp,srp	drp<=srp	.	.	.	.
	NOP	do nothing	.	.	.	.
	ORA reg	A<=A or reg	x	0	x	x
	ORI byte	A<=A or byte	x	0	x	x
	OUT port	data to port<=A	.	.	.	.
	PCHL	PC<=HL	.	.	.	.
	POP rp	rp<=w[SP], SP<=SP+2	.	.	.	.
	PUSH rp	SP<=SP-2, w[SP]<=rp	.	.	.	.
	RAL	rotate A left thru Cf	.	x	.	.
	RAR	rotate A right thru Cf	.	x	.	.
	RC	if Cf=1 then RET	.	.	.	.
**	RDEL	rotate DE left thru Cf	.	x	.	.
	RET	PC<=w[SP], SP<=SP+2	.	.	.	.
8085	RIM	A<=interrupt mask	.	.	.	.
	RLC	rotate A left	.	x	.	.
	RM	if Sf=1 then RET	.	.	.	.
	RNC	if Cf=0 then RET	.	.	.	.
	RNZ	if Zf=0 then RET	.	.	.	.
	RP	if Sf=0 then RET	.	.	.	.
	RPE	if Pf=1 then RET	.	.	.	.
	RPO	if Pf=0 then RET	.	.	.	.
	RRC	rotate A right	.	x	.	.
	RST n	SP<=SP+2, w[SP]<=PC+1, PC<=n*8	.	.	.	.
	RZ	if Zf=1 then RET	.	.	.	.
	SBB reg	A<=A-reg-Cf	x	x	x	x
	SBI byte	A<=A-byte-Cf	x	x	x	x
	SHLD addr	w[addr]<=HL	.	.	.	.
**	SHLI	w[DE]<=HL	.	.	.	.
**	SHLR	shift HL right (extend sign)	.	.	.	.
8085	SIM	interrupt mask<=A	.	.	.	.
	SPHL	SP<=HL	.	.	.	.
	STA addr	b[addr]<=A	.	.	.	.
	STAX B	b[BC]<=A	.	.	.	.
	STAX D	b[DE]<=A	.	.	.	.
	STC	Cf<=1	.	1	.	.
	SUB reg	A<=A-reg	x	x	x	x
	SUI byte	A<=A-byte	x	x	x	x
	XCHG	HL<=DE while DE<=HL	.	.	.	.
	XRA reg	A<=A xor reg	x	0	x	x
	XRI byte	A<=A xor byte	x	0	x	x
	XTHL	HL<=w[SP] while w[SP]<=HL	.	.	.	.

Extended BASIC/Machine Code CALL Command

First, ROM2 must be initialized.

Model 100: `10 CALL 911',,` (apostrophe-comma-comma)

TANDY 200: `10 CALL 921,2',,` (apostrophe-comma-comma)

NEC 8201A: `10 EXEC 1124',,` (apostrophe-comma-comma)

After which, standard ROM or ROM2 routines may be accessed in this way:

Model 100:
```
220 CALL 64902,0,standard rom addr
430 CALL 64902,2,rom2 addr
```

TANDY 200:
```
220 CALL 63622,0,standard rom addr
430 CALL 63622,2,rom2 addr
```

NEC 8201A:
```
220 EXEC 64646,0,standard rom addr
430 EXEC 64646,2,rom2 addr
```

BASIC variables AR%, CF%, ZF%, BC%, DE%, and HL% provide initial register values. Final register values are stored in these variables after the routine has completed.

Accessing ROM2 Routines

First, ROM2 must be initialized.

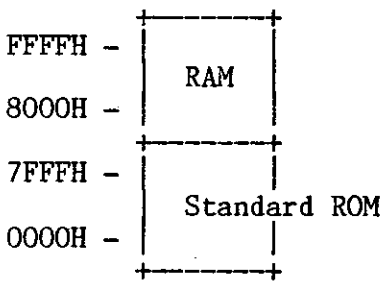
Model 100:	TANDY 200:	NEC 8201A:
MVI A,129	MVI A,130	MVI A,129
CALL 913	CALL 921	CALL 1127

After which, ROM2 routines can be accessed this way:

Model 100:	TANDY 200:	NEC 8201A:
CALL FAA5H	CALL F4D4H	CALL F992H
CALL oprom addr:	CALL oprom addr:	CALL oprom addr:
RST 7	RST 7	RST 7

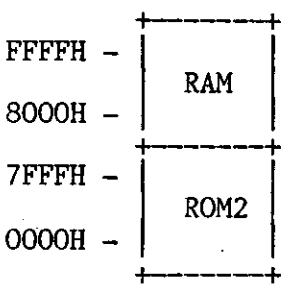
Remember that standard ROM and ROM2 are accessed exclusively of each other. Only one can be referenced at any given time. At the beginning of a BASIC or assembly program memory looks like this:

Standard ROM enabled



After CALL FAA5H (Model 100), CALL F4D4H (TANDY 200) or CALL F992H (NEC 8201A) memory looks like:

ROM2 enabled



When ROM2 is enabled, RST 7 disables it and standard ROM is then accessible. **CAUTION:** A RST 7 must only be done when ROM2 is enabled.

The assembly example on the following page checks for file existance using ROM2 routines. Remember, the assembler can only check for syntax errors, leaving logic errors for debugging. Don't try to run any untested assembly program if your RAM files aren't saved on tape or disk. This rule will protect you from the inevitable bug that trashes RAM or causes a COLD-RESTART that kills all your files.

Model 100/200 Example:

```
Polar Engr & Cons (C)1985
0000:1  ;Assembly example
0000:1
0000:1          ORG 62600                       ;TANDY 200: ORG 60000
F488:1
F488:1  $PROMPT: EQU 0080H
F488:1  $FINDDO: EQU 0089H
F488:1  $PRINT:  EQU 077CH
F488:1  $PRTDEC: EQU 0862H
F488:1  $DIR:    EQU 1B73H
F488:1
F488:1          MVI A,129                       ;TANDY 200: MVI A,130
F48A:1          CALL 913        ;init ROM2      ;TANDY 200: CALL 921
F48D:1
F48D:1  AGAIN:  LXI H,FILEPR
F490:1          CALL FAA5H      ;enable ROM2    ;TANDY 200: CALL F4D4H
F493:1          CALL $PROMPT    ;get user input
F496:1          CPI 1
F498:1          CZ $DIR         ;print file directory
F49B:1          RST 7           ;disable ROM2
F49C:1          JZ AGAIN
F49F:1
F49F:1  ;DE points to input buffer
F49F:1          PUSH D
F4A0:1          LXI H,0
F4A3:1          SHLD FCD2H      ;search all     ;TANDY 200: SHLD F7D2H
F4A6:1          CALL FAA5H      ;enable ROM2    ;TANDY 200: CALL F4D4H
F4A9:1          CALL $FINDDO    ;search directory
F4AC:1          RST 7           ;disable ROM2
F4AD:1          POP D
F4AE:1          XCHG            ;DE points to top of file
F4AF:1                          ;HL points to file name
F4AF:1          RC              ;bad file name
F4B0:1          CALL FAA5H      ;enable ROM2    ;TANDY 200: CALL F4D4H
F4B3:1          CALL $PRINT     ;print file name
F4B6:1          PUSH H
F4B7:1          LXI H,NFMSG
F4BA:1          CZ $PRINT       ;if ZF=1
F4BD:1          LXI H,ATMSG
F4C0:1          CNZ $PRINT      ;if ZF=0
F4C3:1          CNZ $TODECS     ;if ZF=0 print decimal
F4C6:1          RST 7           ;disable ROM2
F4C7:1          POP H
F4C8:1          RET
F4C9:1
F4C9:1  FILEPR: DB 'File name: ',0,1    ;one keyword
F4D6:1          DB 'FILES',13
F4DC:1  NFMSG:  DB ' not found.',13
F4E8:1  ATMSG:  DB ' starts at:',0
F4F4:1          END
0 error(s).
```

NEC 8201A Example:

```
Polar Engr & Cons (C)1985
0000:1  ;Assembly example
0000:1
0000:1          ORG 62088
F288:1
F288:1  $PROMPT: EQU 0080H
F288:1  $FINDDO: EQU 0089H
F288:1  $PRINT:  EQU 077CH
F288:1  $PRTDEC: EQU 0862H
F288:1  $DIR:    EQU 1B73H
F288:1
F288:1          MVI A,129
F28A:1          CALL 1127       ;init ROM2
F28D:1
F28D:1  AGAIN:  LXI H,FILEPR
F290:1          CALL F992H      ;enable ROM2
F293:1          CALL $PROMPT    ;get user input
F296:1          CPI 1
F298:1          CZ $DIR         ;print file directory
F29B:1          RST 7           ;disable ROM2
F29C:1          JZ AGAIN
F29F:1
F29F:1  ;DE points to input buffer
F29F:1          PUSH D
F2A0:1          LXI H,0
F2A3:1          SHLD FBD2H      ;don't exclude any from search
F2A6:1          CALL F992H      ;enable ROM2
F2A9:1          CALL $FINDDO    ;search directory
F2AC:1          RST 7           ;disable ROM2
F2AD:1          POP D
F2AE:1          XCHG            ;DE points to top of file
F2AF:1                          ;HL points to file name
F2AF:1          RC              ;bad file name
F2B0:1          CALL F992H      ;enable ROM2
F2B3:1          CALL $PRINT     ;print file name
F2B6:1          PUSH H
F2B7:1          LXI H,NFMSG
F2BA:1          CZ $PRINT       ;if ZF=1
F2BD:1          LXI H,ATMSG
F2C0:1          CNZ $PRINT      ;if ZF=0
F2C3:1          CNZ $PRTDEC     ;if ZF=0 print decimal
F2C6:1          RST 7           ;disable ROM2
F2C7:1          POP H
F2C8:1          RET
F2C9:1
F2C9:1  FILEPR: DB 'File name: ',0,1    ;one keyword
F2D6:1          DB 'FILES',13
F2DC:1  NFMSG:  DB ' not found.',13
F2E8:1  ATMSG:  DB ' starts at:',0
F2F4:1          END
0 error(s).
```

ROM2 Routine Specifications

The routine descriptions on the following pages explain the register input conditions and output results. Use these descriptions as a guide. Use the debugger on small test programs if further clarification is needed.

Registers:	AR	BC	DE	HL	Cf	Zf
A register (8 bit)	xx					
BC register pair (16 bit)		xx				
DE register pair (16 bit)			xx			
HL register pair (16 bit)				xx		
Carry flag (1 bit)					xx	
Zero flag (1 bit)						xx

Register Usage Key:

** — destroyed (not used as input)
io — input and output
.. — not used (not changed)
in — value used on input (not changed)
ot — result generated as output (not used as input)

Current output device:

If memory location FD85H (Model 100), F885H (TANDY 200) or FC85H (NEC 8201A) is 0 then the current output device is the LCD screen, otherwise it is the parallel printer port.

Arithmetic:

	AR	BC	DE	HL	Cf	Zf

$COMPAR: 0410H (1040) — ** .. in in — ot ot
Cf<=1 if DE<w[HL]
Zf<=1 if DE=w[HL]
Compares the unsigned integer in DE with the unsigned integer w[HL].

$DECPHL: 040AH (1034) — ** .. in in — ot ot
Cf<=1 if DE<HL
Zf<=1 if DE=HL
Compares the unsigned integer in DE with the unsigned integer in HL.

$DEMNHL: 044BH (1099) — ** .. in io — ot **
HL<=DE-HL
Cf<=1 if DE>=HL
Subtracts the unsigned integer in HL from the one in DE.

$INDEX: 0421H (1057) — in .. .. io — .. ..
HL<=w[HL+2*A]
Loads HL through HL+2*A. (A must be in the range 0 to 127.)

$INDEX1: 0853H (2131) — io .. .. io — .. ..
A<=b[HL+A+2]
HL<=w[HL]+b[HL+A+2]
Loads HL through the byte table.

$NEGATE: 0400H (1024) — .. .. .. io — .. ..
HL<=-HL

$OFFSET: 0418H (1048) — in .. .. io — .. ..
HL<=HL+A

String manipulation:

	AR	BC	DE	HL	Cf	Zf

$COPY: 0456H (1110) — io .. io io — ** **
s[HL]<=' ' (for A number of characters)
s[HL]<=s[DE]
DE<=e(s[DE])
HL<=e(s[HL])
Copies the string starting at DE to the address in HL. (Lower case characters translated to upper case.)

$COPYID: 0531H (1329) — .. .. in in — .. ..
i[HL]<=i[DE]
Copies the indentifier starting at DE to the address in HL.

$CPYBLK: 0048H (72) — in .. in in — .. ..
A=>number of bytes to copy
blk[HL]<=blk[DE]
Copies the block starting at DE to the address in HL.

$FILL: 042DH (1069) — in in .. in — .. ..
blk[HL]<=A (for BC number of characters)

$XCHBLK: 0056H (86) — in .. in in — .. ..
A=>number of bytes to exchange
blk[HL]<=blk[DE] while blk[DE]<=blk[HL]
Exchanges the block starting at DE with the block starting at HL.

String compare:	AR	BC	DE	HL	Cf	Zf
$COMPID: 0542H (1346) Cf<=1 if i[DE]<i[HL] Zf<=1 if i[DE]=i[HL]	**	..	in	in	ot	ot
Compares the identifier starting at DE with the one starting at HL.						
$FINDCH: 0450H (1104) HL advanced until b[HL]=A	in	..	..	io	0	1
Searches starting at HL for the value in A.						
$FINDEN: 0525H (1317) HL<=e(i[HL])	..	..	..	io	..	..
Advances HL to first character past the identifier that starts at HL.						
$FINDID: 0498H (1176) Advance HL to first "non-blank" Zf<=1 if end-of-line reached	**	..	..	io	0	ot
Advances past no more than one comma. End-of-line characters are any one of: null (ASCII 0), cr (ASCII 13), eof (ASCII 26), or semi-colon (ASCII 59). Tabs (ASCII 9) are treated like blanks.						
$FINDNS: 04BCH (1212) Advance HL to first "non-blank" Zf<=1 if end-of-line reached	**	..	..	io	0	ot
Functions exactly like $FINDID: except that HL is incremented by one before the first test, and commas are always "non-blank".						
$FINDSP: 0489H (1161) Advance HL to first "space" Zf<=1 if end-of-word reached	**	..	..	io	**	ot
A "space" is either a blank (ASCII 32), a comma (ASCII 44), or a semi-colon (ASCII 59). End-of-word occurs if the "space" character reached is a comma or a semi-colon.						
$LEN: 043BH (1083) BC<=length of n[HL]	..	ot	..	in	..	..
A null (ASCII 0) terminates the string.						
$NXTLIN: 04C3H (1219) HL advanced to next line Zf<=1 if end-of-file reached	**	..	..	io	**	ot
Advances to the character right after the end-of-line characters (cr lf). When an eof (ASCII 26) is encountered, the Zero flag is set.						
$NXTWRD: 04D2H (1234) HL advanced to next word Zf<=1 if end-of-line reached	**	..	..	io	0	ot
Advances to the next word by calling $FINDSP followed by $FINDID. See those routines for more information.						

String conversion:	AR	BC	DE	HL	Cf	Zf

$FRANYS: 0719H (1817) ** .. ot in ot 0
DE<=value of constant in s[HL]
Cf<=1 if invalid constant
Calculates the value of s[HL] by evaluating the constant in the string s[HL]. This constant may be decimal, hexadecimal, octal, or binary.

$FRBINS: 06BDH (1725) ** .. ot in ot 0
DE<=binary value of s[HL]
Cf<=1 if invalid binary value
The last character in the string s[HL] may be a B.

$FRDECS: 0675H (1653) ** .. ot in ot 0
DE<=decimal value of s[HL]
Cf<=1 if invalid decimal value
The last character in the string s[HL] may be a D.

$FREXPS: 00ADH (173) ** .. ot in ot 0
DE<=value of expression n[HL]
Cf<=1 if invalid expression
Calculates the value of n[HL] (terminated by a null, cr, semi-colon, or eof) by evaluating the algebraic expression in the string n[HL]. This expression may contain decimal, hexadecimal, octal, or binary constants. The addition, subtraction, multiplication, division, and shift operators as well as parentheses are allowed.

$FRHEXS: 0646H (1606) ** .. ot in ot 0
DE<=hexadecimal value of s[HL]
Cf<=1 if invalid hexadecimal value
The last character in the string s[HL] may be an H.

$FROCTS: 06E9H (1769) ** .. ot in ot 0
DE<=octal value of s[HL]
Cf<=1 if invalid octal value
The last character in the string s[HL] may be an O.

$TOBINS: 05DEH (1502) .. .. in in .. ..
s[HL]<=binary string form of DE
When D<>0, the binary string gets 16 digits, otherwise it gets 8 digits.

$TODECS: 0596H (1430) .. .. in in .. ..
s[HL]<=decimal string form of DE
All leading zeros are suppressed.

$TOHEXS: 056BH (1387) .. .. in in .. ..
s[HL]<=hexadecimal string form of DE
When D<>0, the hexadecimal string gets 4 digits, otherwise it gets 2 digits.

$TOOCTS: 05FFH (1535) .. .. in in .. ..
s[HL]<=octal string form of DE
When DE>511, the octal string gets 6 digits, otherwise it gets 3 digits.

$TO4HXS: 0625H (1573) .. .. in in .. ..
s[HL]<=hexadecimal string form of DE
Always 4 digits in the string.

LCD and printer output: AR BC DE HL Cf Zf

(The routine $BRKCHK is called by all of these routines.)

$LCDOUT: 075EH (1886) .. .. .. in .. ..
LCD screen<=s[HL]+next character
Sends the string starting at HL plus its non-string terminating character to the LCD screen.

$LPTOUT: 0763H (1891) .. .. .. in .. ..
printer<=s[HL]+next character
Sends the string starting at HL plus its non-string terminating character to the printer.

$OUTCHR: 0794H (1940) in .. ..
LCD screen or print<=A
Sends the character in A to the current output device. Carriage returns are printed with line-feeds.

$PRINT: 077CH (1916) .. .. .. in .. ..
LCD screen or printer<=s[HL]+next character
Sends the string starting at HL plus its non-string terminating character to the current output device.

$PRTCHR: 069DH (1693) in .. .. .* ** **
LCD screen or print<=A
Sends the character in A to the current output device. When the character is unprintable, print a dot instead.

$PRTCR: 0825H (2085) .. ..
Sends a cr (ASCII 13) to the current output device.

$PRTDEC: 0862H (2146) .. .. in .. .. ..
LCD screen or print<=DE in decimal string form
Sends the decimal encoding to the current output device. Leading zeros are suppressed.

$PRTDOT: 081FH (2079) .. ..
Sends a period (ASCII 46) to the current output device.

$PRTHEX: 086CH (2156) .. .. in .. .. ..
LCD screen or print<=DE in 2 or 4 digit hexadecimal string form
Sends the two or four digit hexadecimal encoding to the current output device. See $TOHEXS.

$PRTSP: 0819H (2073) .. ..
Sends a space (ASCII 32) to the current output device.

$PRTTAB: 0813H (2067) .. ..
Sends a tab (ASCII 9) to the current output device.

$PRT4HX: 0876H (2166) .. .. in .. .. ..
LCD screen or print<=DE in 4 digit hexadecimal string form
Sends the four digit hexadecimal encoding to the current output device.

Keyboard input:

	AR	BC	DE	HL	Cf	Zf
$BUFTST: 00B0H (176)	**	..	..	io	0	ot

HL<=advanced to next input word
Zf<=set if buffer is empty
Checks for pending input in the buffer.

	AR	BC	DE	HL	Cf	Zf
$GETVAL: 0095H (149)	**	..	io	io	io	1

s[DE]=>prompt string
DE<=destroyed
HL=>default user value
HL<=user value
Cf=>0:prompt only if input buffer is empty
Cf=>1:always prompt for new input
Cf<=1 if bad expression entered
The string starting at DE must be terminated by two zero byte values.
Example prompt string

```
DB 'Enter a value: ',0,0
```

When the user enters just a cr or a comma, the HL value input to the routine is returned as the user value.

	AR	BC	DE	HL	Cf	Zf
$PROMPT: 0080H (128)	ot	..	ot	io	**	ot

A<=index of keyword matched with user input
DE<=address of user input (input buffer)
s[HL]=>prompt and keyword string
HL<=destroyed
Zf<=1 if no keyword was matched
Here is a sample prompt and keyword string.

```
DB 'Okay? ',0,2,'YES NO',13
```

If the user enters YES then the keyword index returned in A would be 2. If NO was entered then A would be 1 and if neither was entered then A would be 0. The general form of the prompt and keyword string is:

```
DB 'prompt',0,n,'keyw_n ... keyw_2 keyw_1',13
```

A single blank is used to separate keywords and the final byte is always 13. This routine lists the keywords for the user when a ? is entered, if 'n' is not zero.

	AR	BC	DE	HL	Cf	Zf
$SETINB: 00A1H (161)	..	..	in	..	..	..

input buffer<=s[DE]
Assigns the string starting at DE to the input buffer. The next time input is retrieved by a $PROMPT routine the user won't be asked for any input because the input buffer already has some.

Files (.DO):

	AR	BC	DE	HL	Cf	Zf
$COMPDO: 1B79H (7033)	**	ot	ot	**	ot	ot

BC<=number of bytes that were the same
DE<=number of whole lines matched
Cf<=1 if files are different
Zf<=1 if user aborted compare
Prompts the user for two files to compare.

	AR	BC	DE	HL	Cf	Zf
$COPYDO: 1B76H (7030)	**	**	**	**	ot	**

Cf<=1 if out-of-memory
Prompts the user for a file to copy and a new file to copy into.

	AR	BC	DE	HL	Cf	Zf
$DIR: 1B73H (7027)	..	..	..	..	..	..

Prints the directory of RAM files with sizes to the current output device.

Files (.DO): continued

	AR	BC	DE	HL	Cf	Zf
$FILLEN: 008CH (140) DE<=file length HL=>directory address	**	..	ot	in	**	**
$FINDDO: 0089H (137) blk[DE]=>file name DE<=file's directory address HL<=file's top address Cf<=1 if file name is invalid Zf<=1 if file is not found	**	..	io	ot	ot	ot

Searches the directory for a .DO file. The word at location FCD2H (Model 100), F7D2H (TANDY 200) or FBD2H (NEC 8201A) contains the directory address excluded from the search. If ZF=1 then HL will be zero.

$INNAME: 0083H (131) DE=>exclude this directory address DE<=input file's directory address HL<=input file's top address Zf<=1 if user only typed ENTER (no input file)	**	..	io	ot	**	ot

Prompts the user for an input file name.

$KILLDO: 00A4H (164) HL=>file's directory address	..	..	..	in	..	..
$MAKHOL: 008FH (143) BC=>number of bytes to insert HL=>insert gap in file starting at this address Cf<=1 if out-of-memory	**	in	..	in	ot	**
$MASDEL: 00AAH (170) BC=>number of bytes to delete HL=>start deleting in file at this address	**	in	..	in	**	0
$OUTNAM: 0086H (134) DE=>exclude this directory address DE<=output file's directory address HL<=output file's top address Zf<=1 if user only typed ENTER (no output file)	**	..	io	ot	**	ot

Prompts the user for an output file name. When the file already exists, the user is prompted for overwrite permission. If Zf=0 then the file created or overwritten is null.

$SUBS: 1B7CH (7036) BC=>pattern length n[DE]=>replacement string HL=>file top HL<=file end CF<=set if out of memory during substitution	**	in	in	io	**	ot

Replaces all occurrences of the pattern string with the replacement string in the file. The memory word at location FD2EH (Model 100), or F82EH (TANDY 200) or FC2EH (NEC 8201A) contains the pattern string's start address.

$WRITES: 0092H (146) n[DE]=>string to write HL=>start writing string at this address HL<=one past last address written	..	..	in	io	..	..

Writes a string into a file. This routine does not insert a gap, it overwrites existing characters.

Misc: AR BC DE HL Cf Zf

$BITTST: 04D8H (1243) io .. .. in ot 0
A=>bit index to be tested
A<=is destroyed
HL=>bit string starts at this address
Cf<=1 if the tested bit is zero
A bit string is a sequence of bytes. Each byte contains 8 bits and the high order bits are indexed first. For example
DB 01000100B,10000001B
Bits at indices 1, 5, 8, and 15 are ones.

$BRKCHK: 0735H (1845) .. ..
Checks for BREAK key or PAUSE key (Model 100/200) or STOP key or CTRL-S (NEC 8201A). When the PAUSE key or CTRL-S is hit, wait for another before continuing. When the BREAK key or STOP key is hit, call the break handler. The default break handler calls the main menu. To override this, store the desired handler (subroutine) entry address like so

Model 100:	TANDY 200:	NEC 8201A:
MVI A,CFH	MVI A,CFH	MVI A,CFH
STA FDBEH	STA F8BEH	STA FCBEH
LXI H,handler_addr	LXI H,handler_addr	LXI H,handler_addr
SHLD FDBFH	SHLD F8BFH	SHLD FCBFH

The subroutine at 'break_handler_entry_addr' is called if the BREAK key or STOP key has been hit before $BRKCHK is called. When debugging a program that defines its own break handler, DBG's break handler is still used. To simulate a BREAK key or STOP key being hit: 1) Hit the BREAK or STOP key. 2) Call the break handler from the debugger.

$HEXTST: 0638H (1592) io .. .. .. ot **
A<=A-'0' or A<=A-'A'+10
Cf<=1 if A is not one of: **0123456789ABCDEF**
Checks if character in A represents a hexadecimal digit.

$IDCHAR: 04FDH (1277) io .. .. .. ot 0
A<=uppercase of A
Cf<=1 if A is not one of: **$0..9?@A..Z_a..z**
Checks if character in A is a valid identifier character.

$MON: 0850H (2128) .. ..
Invokes DBG on the instruction immediately following the call to $MON. A RST 7 must not follow the call to $MON. (This call should not be simulated.)

$SHIFT: 009BH (155) .. .. in .. in ..
DE=>insert/delete one byte at this address
Cf=>1:insert, Cf=>0:delete
Inserts or deletes bytes in BASIC programs. When this routine is called, $SHIFTE must be called once at the very end of the assembly program to do the final clean-up.

$SHIFTE: 009EH (158) .. ..
End of program cleanup for $SHIFT. Must be called at the end of the assembly program, if $SHIFT has ever been called. This routine also fixes the RAM file directory when running machine code combined with BASIC programs.

Appendix A: **Diagnostic Program**

This diagnostic program calculates the check sum of your ROM2. If you are having trouble with your ROM2 enter this BASIC program and run it. If the check sum printed does not match the number on the label of your ROM2 then it may be defective. If you think you have a defective ROM2 please call us so we can help correct the problem.

Model 100 diagnostic program:

```
10 CLEAR 256,HIMEM-33
20 FOR I=HIMEM TO HIMEM+32:READ X:T=T+X:NEXT
30 IF T<>3651 THEN 150
40 RESTORE
50 FOR I=HIMEM TO HIMEM+32:READ X:POKE I,X:NEXT
60 POKE HIMEM+23,(HIMEM+11-32768) MOD 256
70 POKE HIMEM+24,(HIMEM+11)/256
80 IF PEEK(HIMEM+23)+PEEK(HIMEM+24)*256<>HIMEM+11 THEN 150
90 CALL HIMEM
100 GOTO 160
110 DATA 243,17,0,0,33,0,0,62,1,211,224
120 DATA 26,133,111,124,206,0,103,19,122
130 DATA 254,32,194,0,0,175,211,224,251,205
140 DATA 212,57,201
150 BEEP:PRINT"Program entered incorrectly."
160 CLEAR 256,HIMEM+33
```

TANDY 200 diagnostic program: (from BANK #1 only)

```
10 CLEAR 256,HIMEM-33
20 FOR I=HIMEM TO HIMEM+32:READ X:T=T+X:NEXT
30 IF T<>3449 THEN 150
40 RESTORE
50 FOR I=HIMEM TO HIMEM+32:READ X:POKE I,X:NEXT
60 POKE HIMEM+23,(HIMEM+11-32768) MOD 256
70 POKE HIMEM+24,(HIMEM+11)/256
80 IF PEEK(HIMEM+23)+PEEK(HIMEM+24)*256<>HIMEM+11 THEN 150
90 CALL HIMEM
100 GOTO 160
110 DATA 243,17,0,0,33,0,0,62,2,211,216
120 DATA 26,133,111,124,206,0,103,19,122
130 DATA 254,32,194,0,0,175,211,216,251,205
140 DATA 11,71,201
150 BEEP:PRINT"Program entered incorrectly."
160 CLEAR 256,HIMEM+33
```

NEC 8201A diagnostic program: (from BANK #1 only)

```
10 CLEAR 256,62200
20 FOR I=62200 TO 62236:READ X:T=T+X:NEXT
30 IF T<>4554 THEN 120
40 RESTORE
50 FOR I=62200 TO 62236:READ X:POKE I,X:NEXT
60 EXEC 62200
70 PRINT PEEK(63912)+PEEK(63913)*256
80 END
90 DATA 243,17,0,0,33,0,0,219,161,230,12,246,1,211,161
100 DATA 26,133,111,124,206,0,103,19,122
110 DATA 254,32,194,7,243,219,161,230,12,211,161,251,201
120 BEEP:PRINT"Program entered incorrectly."
```

Appendix B: **ASM Error Table**

This table explains the various error combinations that can occur when assembling a program. During pass 1 the assembler does not print source lines, so errors found during that pass have no source line context. During pass 2 the source line is printed immediately prior to the error message.

Pass 1 Errors

In use:X:ABCD
Reason: Multiply defined label 'X:'. Label 'X:' is used more than once as a statement label.
Fix: Change the other 'X:' labels to different names.

In use:X:$
No label:X:$
Reason: Forward label reference to a MAC statement. The statement '&X:' is used to invoke macro 'X:' before the macro has been defined.
Fix: Put macro definition 'X:' before the invocation statement '&X:'.

In use:X:$BCD
No label:X:$BCD
Reason: Label 'X:' is both multiply defined and forward referenced.
Fix: Change the other 'X:' labels and remove forward references.

No label:X:
Reason: Label 'X:' is undefined.
Fix: use 'X:' as a statement label or correct the spelling.

No label:X:$BCD
Reason: Forward label reference in an EQU statement. Label 'X:' is used in a EQU statement's expression before it is used as a statement label.
Fix: Put the statement label 'X:' before the EQU expression that uses 'X:'.

Bad op.
Out of memory.
Reason: ORG or DS expression is bad or has a forward reference.
Fix: Make all ORG and DS expressions correct, and remove all forward references.

Out of memory.
When: If ?FRE(0) is less than 256 or _ASM_.DO is a null file.
Reason: No more RAM space is available for label table.
Fix: Free up some RAM space by deleting files or changing HIMEM before reassembling.

Out of memory.
When: If ?FRE(0) is greater than 256 and _ASM_.DO is not a null file.
Reason: No more macro expansion space is available or infinitely recursive include file.
Fix: Remove the deeply nested macro reference or the infinitely recursive file include. The first part of the label table file '_ASM_.DO' will contain the failing expansion, so refer to it for further assistance.

Appendix B: **ASM Error Table** (continued)

Pass 2 Errors

&file
Bad file.
Reason: Include 'file'.DO not found in RAM directory.
Fix: Correct spelling of 'file' or read in 'file' from tape or disk.

EQU expr
Bad op.
Reason: An EQU statement must have a statement label.
Fix: Add a statement label.

MAC
Bad op.
Reason: A MAC statement must have a statement label.
Fix: Add a statement label.

&X:
Bad op.
Reason: No definition of macro 'X:'.
Fix: Add definition of macro 'X:'.
Reason: Macro 'X:' contains a macro definition.
Fix: Do not nest macro definitions.

garbage
Bad op.
Reason: 'garbage' is not an 8085 instruction or an assembler pseudo op.
Fix: Correct the spelling of 'garbage'.

opcode,expr
Bad arg.
Reason: 'expr' is not a valid algebraic expression or is out of the valid range for 'opcode'.
Fix: Correct 'expr' so that it is a valid.

opcode garbage
Line too long.
Reason: The text 'garbage' is spurious text.
Fix: Add a missing comment delimiter or correct the spelling of 'opcode'.

opcode
No arg.
Reason: 'opcode' is missing one or two arguments.
Fix: Add the needed args.

opcode,expr
No arg.
Reason: 'opcode' is missing its second argument.
Fix: Add the second arg.

opcode
PC bad.
Out of memory.
Reason: Program counter is outside of the HIMEM to MAXRAM-1 range.
Fix: Either move the program's entry address (by changing the ORG expression), or change HIMEM with the BASIC CLEAR command.

Appendix C: **Macros and Ifs**

Recursive Macros

Combining IF statements and macro statements allows recursive macros to be defined. Each time a macro is invoked from within a macro the current macro is suspended and the invoked macro is processed. This locks up a certain amount macro expansion space. For this reason nested macro substitution may not be possible because of insufficient macro expansion space. This is a hard limit and may prevent certain complex recursive (or even non-recursive, but heavily nested) macros from being expanded. Here is an example of a recursive macro that works, provided the input args are not too many characters long.

```
BITMASK: MAC
         IF #2-1
         &BITMASK: #1;#2-1
         END
         DB (#1).(#2)
         END
```

Invoking the macro like this:

```
         &BITMASK: 1;7
```

is the same as writing this assembly line:

```
         DB 1,2,4,8,16,32,64,128
```

Useful IF Statements

The first column shows the situation under which code generation is desired. The second shows the form of the IF statement needed to obtain that result. X, Y, and Z represent the expressions to be tested.

x>=0	IF x
x<=0	IF -x
x>0	IF x-1
x<0	IF -x-1
x=0	IFZ x
x<>0	IFNZ x
x mod 2=0	IFZ x.15
x mod 2=1	IFNZ x.15
x>=y	IF x-y
x>y	IF x-y-1
x=y	IFZ x-y
x<>y	IFNZ x-y
x mod y=z	IFZ x-(x/y)*y-z
x mod y<>z	IFNZ x-(x/y)*y-z
x>=y and x<=z	IF x-y,z-x (requires two ENDs)
x>=y and x<=z	IFZ (y-x).-15 + (x-z).-15
x<y or x>z	IFNZ (x-y).-15 + (z-x).-15

Appendix D: **Example Macros**

```
SAVE:   MAC             ;save all the registers on the stack
          PUSH PSW
          PUSH B
          PUSH D
          PUSH H
          END

        &SAVE:          ;all registers saved on the stack

RESTORE: MAC            ;restore all registers from the stack
           POP H
           POP D
           POP B
           POP PSW
           END

        &RESTORE:       ;all registers restored from the stack

PUSHI:  MAC             ;push word onto stack
          PUSH H        ;save HL
          LXI H,#1      ;HL gets #1
          XTHL          ;HL restored, #1 pushed
          END

        &PUSHI: 44B4H   ;push 44B4H onto the stack

ROM2:   MAC             ;call a ROM2 routine
          CALL FAA5H    ;enable ROM2 (TANDY 200: F4D4H, NEC 8201A: F992H)
          CALL #1       ;call ROM2 routine
          IFNZ #1-0850H ;$MON starts simulating with the standard
                        ;ROM enabled, so don't do a RST 7 for it
            RST 7       ;enable the standard ROM
            END
          END

        &ROM2: 0850H    ;call $MON

LDAX:   MAC             ;load A with b[BC], b[DE], or b[HL]
          IFZ '#1'-'B'
            LDAX B      ;A<=b[BC]
            ELSE
            IFZ '#1'-'D'
              LDAX D    ;A<=b[DE]
              ELSE
              IFZ '#1'-'H'
                MOV A,M ;A<=b[HL]
                ELSE
        !&LDAX: #0      ;#1 is bad, flag as a 'Bad op.' error
                END
              END
            END
          END

        &LDAX: B;       ;A<=b[BC]
        &LDAX: H;       ;A<=b[HL]
        &LDAX: L;       ;error flagged by macro
```

Appendix D: **Example Macros** (continued)

```
STI:    MAC             ;store byte in memory
          PUSH PSW      ;save AF
          IFZ #2
            XRA A       ;A<=0
            ELSE
            MVI A,#2    ;A<=#2
            END
          STA #1        ;b[#1]<=A
          POP PSW       ;AF restored
          END

        &STI: COUNT:;0  ;b[COUNT:]<=0
        &STI: FLAG:;1   ;b[FLAG:]<=1

SID:    MAC             ;store word in memory
          PUSH H        ;save HL
          LXI H,#1      ;HL<=#1
          SHLD #2       ;w[HL]<=#2
          POP H         ;HL restored
          END

        &SID: TEN:;10   ;w[ADDR:]<=10

SID2:   MAC             ;store word in memory (like SID:)
          &STI: #1;(#2).8.-8    ;b[#1]<=low byte of #2
          &STI: #1+1;(#2).-8    ;b[#1+1]<=high byte of #2
          END

REP:    MAC             ;repeat
          IF #1-1
            #2
            &REP: -1+#0
            END
          END

        &REP: 4;INX H   ;increment HL 4 times

MULT:   MAC             ;multiple statements on one line
          #1
          #2
          #3
          #4
          #5
          #6
          #7
          #8
          END #9        ;error if too many statements

        &MULT: INX H;MOV A,M;DCX H

MREP:   MAC             ;repeat multiple statements
          IF #1-1
            &MULT: #2;#3;#4;#5;#6;#7;#8;#9
            &MREP: -1+#0
            END
          END

        &MREP: 2;MOV A,M;CPI 'A';JZ FOUND:;INX H
```

Appendix E: **ASM/DBG Label Table Format**

The assembler creates the file _ASM_.DO with this format:

CR
label 1 : 4digit hex **CR**
label 2 : 4digit hex **CR**
.
.
.
label n : 4digit hex **CR**
> **EOF**

Each statement label in the last program assembled has one line in this file. The labels are organized alphabetically. The value of a macro label is the address of the macro definition in RAM at the time of the assembly and may be meaningless after the assembly is completed. This table is used by DBG for symbol resolution.

Assembler Output File

When the assembler output file option is used, the assembled machine code is appended to the end of the output file specified. The output file has the same initial format as _ASM_.DO. The machine codes are appended to the file two hex characters for each byte. This is the format:

CR
label 1 : 4digit hex **CR**
label 2 : 4digit hex **CR**
.
.
.
label n : 4digit hex **CR**
> 2digit hex 2digit hex ... 2digit hex **EOF**

Spaces (ASCII 32) may exist between the last 2digit hex and **EOF.**

Appendix F: **Example Break Handler**

The routine $BRKCHK and other printing routines (LCD and PRT) check to see if the BREAK key (Model 100/200) or STOP key (NEC 8201A) has been hit. If you want your assembly program to do something special (rather than return to the main MENU) you can override the default break handler with one of your own. The following program establishes a break handler that causes the program to return to BASIC when the BREAK key or STOP key is hit.

Model 100/200 example:

```
        ORG 57000

ROM2:   MAC
        CALL FAA5H      ;enable ROM2            ;TANDY 200: CALL F4D4H
        CALL #1         ;call ROM2 routine
        IFNZ #1-0850H
        RST 7           ;disable ROM2
        END
        END

LCDOUT: EQU 075EH

        MVI A,129                               ;TANDY 200: MVI A,130
        CALL 913        ;initialize ROM2        ;TANDY 200: CALL 921

        CALL PROG

        LXI H,STOP
        &ROM2: LCDOUT
        RET

STOP:   DB 'Stop.',13


PROG:   LXI H,0
        DAD SP
        SHLD STKPTR     ;save the entry stack level

        MVI A,CFH
        STA FDBEH                               ;TANDY 200: STA F8BEH
        LXI H,BREAK
        SHLD FDBFH                              ;TANDY 200: SHLD F8DFH
;default break handler replaced

        LXI H,RUNNING
LOOP:   &ROM2: LCDOUT
        JMP LOOP

RUNNING: DB 'Running.',0


;BREAK handler
BREAK:  LHLD STKPTR
        SPHL            ;return to entry stack level
        RET             ;return out of PROG

STKPTR: DS 2            ;allocate a word for storage
        END
```

Appendix F: **Example Break Handler (continued)**

NEC 8201A example:

```
        ORG 57000

ROM2:   MAC
        CALL F992H      ;enable ROM2
        CALL #1         ;call ROM2 routine
        IFNZ #1-0850H
        RST 7           ;disable ROM2
        END
        END

LCDOUT: EQU 075EH

        MVI A,129
        CALL 1127       ;initialize ROM2

        CALL PROG

        LXI H,STOP
        &ROM2: LCDOUT
        RET

STOP:   DB 'Stop.',13


PROG:   LXI H,0
        DAD SP
        SHLD STKPTR     ;save the entry stack level

        MVI A,CFH
        STA FCBEH
        LXI H,BREAK
        SHLD FCBFH      ;override default break handler

        LXI H,RUNNING
LOOP:   &ROM2: LCDOUT
        JMP LOOP

RUNNING: DB 'Running.',0


;BREAK handler
BREAK:  LHLD STKPTR
        SPHL            ;return to entry stack level
        RET             ;return out of PROG

STKPTR: DS 2            ;allocate a word for storage
        END
```

Appendix G: **Notation**

Syntax

Notation:	Explanation
a ::= b	construct 'a' is defined as construct 'b'
[a]	use zero or one of 'a'
[a]*	use zero or n of of 'a' repeatedly
a | b	use either 'a' or 'b' once
x..y	select a value in the range x to value y (inclusive)
id	non-terminal syntax construct
c	terminal syntax char

Example:
Dec ::= [**0..9**]* [**D**]
means that a decimal constant, Dec, is defined as any sequence of digits, followed by an optional terminal syntax char **D.**

Registers, Register Pairs, and Flags

x<=expression	after completing the instruction (or routine) register, register pair, or flag 'x' contains the value of 'expression'
x=>y	before initiating the instruction (or routine) register, register pair, or flag 'x' must contain input data 'y'

Memory References

b[x]	byte value in memory location x
blk[x]	memory block beginning at address x
i[x]	identifier string beginning at address x
n[x]	null terminated string beginning at address x
s[x]	text string beginning at address x
w[x]	word value in memory location x and x+1
e(i[x])	address of non-identifier char that ends i[x]
e(n[x])	address of null char that ends n[x]
e(s[x])	address of non-string char that ends s[x]

Char Definitions

Term	Definition
char	an ASCII character (one byte)
id char	ASCII $, 0..9, ?, @, A..Z, _, a..z
null char	ASCII value 0
string char	ASCII values 9, 32..127

Appendix H: **Command Syntax**

CMD> one line commands.....

BREAK or **STOP**	ASMN file [[file|/]] **ENTER**
ENTER	COPY file file [[Y|N]] **ENTER**
? **ENTER**	DBG **ENTER**
ASM file [[file|/]] **ENTER**	FEQ file file **ENTER**
ASML file [[file|/]] **ENTER**	FILES **ENTER**
ASMLN file [[file|/]] **ENTER**	RN [,num [,num [,num [,num]]]] **ENTER**

CMD> multiple line commands.....

ASM **ENTER**	**File:** file [[file|/]] **ENTER**
ASML **ENTER**	**File:** file [[file|/]] **ENTER**
ASMLN **ENTER**	**File:** file [[file|/]] **ENTER**
ASMN **ENTER**	**File:** file [[file|/]] **ENTER**
CH **ENTER**	**File:** file **ENTER**
	from: pattern string **ENTER**
	to: replacement string **ENTER**
COPY **ENTER**	**File:** file **ENTER**
	File: file **ENTER**
	[**Rewrite:** [Y|N] **ENTER**]
FEQ **ENTER**	**File:** file **ENTER**
	File: file **ENTER**
RN **ENTER**	**F,L,T,S:** [num [,num [,num [,num]]]] **ENTER**

num::= [**BREAK** or **STOP** | null | expr]

File: commands.....

BREAK or **STOP**	? **ENTER**
ENTER	FILES **ENTER**
RAM .DO file **ENTER**	MENU **ENTER**

DBG> commands.....

BREAK or **STOP**	IN [expr] **ENTER**
ENTER	L [expr] **ENTER**
expr **ENTER**	LBS **ENTER**
[expr,]8085 instruction **ENTER**	LCD **ENTER**
expr,DB expr [,expr]* **ENTER**	LPT **ENTER**
expr,DS expr **ENTER**	MAP [expr] **ENTER**
expr,DW expr [,expr]* **ENTER**	MENU **ENTER**
? **ENTER**	N [expr] **ENTER**
CB **ENTER**	OUT [expr] **ENTER**
CBS **ENTER**	RUN **ENTER**
CE [expr] **ENTER**	S [expr] **ENTER**
D **ENTER**	SE [expr] **ENTER**
EX [expr] **ENTER**	SKP **ENTER**
GE [expr] **ENTER**	TN **ENTER**
GO [expr] **ENTER**	TY **ENTER**

ASM statements.....

EOF	[label:] IF expr [,expr]* **CR**
[label:] **CR**	[label:] IFNZ expr [,expr]* **CR**
[label:] 8085 instruction **CR**	[label:] IFZ expr [,expr]* **CR**
[label:] DB expr [,expr]* **CR**	label: MAC **CR**
[label:] DS expr **CR**	[label:] ORG expr **CR**
[label:] DW expr [,expr]* **CR**	[label:] &RAM .DO file **CR**
[label:] ELSE **CR**	[label:] &macro label: [arg [;arg]*] **CR**
[label:] END **CR**	[label:] &&RAM .DO file **CR**
label: EQU expr **CR**	[label:] &&macro label: [arg [;arg]*] **CR**

Appendix I: **Expression Syntax**

All expressions are processed from left to right. The following is a BNF definition of the expression syntax. The expressions allowed by ROM2 are essentially the same as those allowed by BASIC.

BNF syntax defintion:

```
expression ::= primary [ operator primary ]*
primary ::= [ - ]* value
value ::= $ | PC | constant | label [ : ] | ( expression )
constant ::= hex H | dec [ D ] | oct O | bin B | ' ascii '
hex ::= [ 0..9 | A..F | a..f ]* (range: 0..FFFFH)
dec ::= [ 0..9 ]*               (range: 0..65535D)
oct ::= [ 0..7 ]*               (range: 0..1777770)
bin ::= [ 0..1 ]*               (range: 0..1111111111111111B)
ascii ::= asciichar [ asciichar ]
asciichar ::= ascii values: 9 | 32..38 | 40..126 | 128..255
label ::= [ $ | 0..9 | ? | @ | A..Z | _ | a..z ]*9
operator ::= + | - | * | / | .
```

Operator precedence:

```
highest : unary negation
middle  : multiplication, division, and shift
lowest  : addition and subtraction
```

Operator evaluation (valid result range):

```
-x : unary negation (no overflow possible)
x+y: signed addition (range: -32768..32767)
x-y: signed subtraction (range: -32768..32767)
x*y: signed multiplication (range: -32768..32767)
x/y: signed division (range: -32768..32767)
     (divide by zero yields overflow)
x.y: shift (no overflow possible, y range: -16..16)
```

Shift operator definition (x.y):
If y>0 then shift x y bits left else shift x -y bits right. Zero bits are shifted in. (The low 16 bits of the integer formula x*(2**y) is an equivalent definition.) Example shift operations:

```
ABCDH.-8 yields ABH
ABCDH.8 yields CDOOH
ABCDH.8.-8 yields CDH
ABCDH.-8.8 yields ABOOH
21.2 yields 84
21.-1 yields 10
```

Appendix J: **DBG Simulation Examples** (Model 100)

These example command sequences operate on this program. Use TEXT to enter it into a file named PROG.DO. The ORG value can be changed to a different value, if 62900 is not suitable.

```
          ORG 62900

PROG:     LXI D,-2          ;entry to main program
          LXI H,2
          CALL B
B_DONE:   DAD D
EXIT:     RET

B:        MVI B,0           ;entry to subroutine B
LOOP_B:   CALL C            ;loop 10 times on five instructions
          INR B
          MOV A,B
          CPI 10
          JNZ LOOP_B
          RET

C:        MVI C,0           ;entry to subroutine C
LOOP_C:   INR C             ;loop 248 times on four instructions
          MOV A,C
          CPI 248
          JNZ LOOP_C
          RET
```

Enter BASIC and allocate user high memory area.

```
Ok
CLEAR 256,62900 (use your ORG value, if different)
```

Assemble the program.

```
Ok
(function key 6)
CMD> ASMN PROG
Polar Engr & Cons (c)1985
0 error(s).
```

Enter DBG.

```
Ok
(function key 6)
CMD> DBG
Polar Engr & Cons (c)1985
FD8C CALL FAA5    Z NC PE P SP=F160.1E20
 A=04 BC=00BE.50 DE=0000.C3 HL=0850.62
DBG> DBG command
```

Each of the following examples assume that you have just entered DBG.

```
1) CALL PROG          * immediate assembly command executes PROG,
                        returns to FD8CH
2) CE PROG            * stacks FD8CH, jumps to PROG
   OUT                  simulates PROG, executes B, returns to FD8CH
```

Appendix J: **DBG Simulation Examples** (Model 100--continued)

```
 3) CE PROG          * stacks FD8CH, jumps to PROG
    N 3                simulates three instructions, stops at B
 4) CE PROG          * stacks FD8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
 5) CE PROG          * stacks FD8CH, jumps to PROG
    1 B                set break point 1 at B
    GE                 simulates three instructions, stops at B
 6) CE PROG          * stacks FD8CH, jumps to PROG
    1 B                set break point 1 at B
    GO                 simulates three instructions, stops at B
                       (DBG simulates the jump part of CALL B and
                       encounters the break point)
-------------------------------------------------------------------
 7) CE PROG          * stacks FD8CH, jumps to PROG
    S 3                simulates two instructions, executes B, (three
                       total) stops at B_DONE
 8) CE PROG          * stacks FD8CH, jumps to PROG
    SE 3               simulates two instructions, simulates B and C,
                       (three total) stops at B_DONE (takes 10 seconds)
 9) CE PROG          * stacks FD8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    OUT                simulates B, executes C 10 times, stops at B_DONE
10) CE PROG          * stacks FD8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    S 5*10+2           simulates 42 instructions, executes C 10 times,
                       (52 steps) stops at B_DONE
11) CE PROG          * stacks FD8CH, jumps to PROG
    N 3                simulates three instructions, stops at B
    N 2+10*(5+2+248*4) simulates 9992 instructions, stops at B_DONE
                       (takes 10 seconds)
12) CE PROG          * stacks FD8CH, jumps to PROG
    IN 2+10*2          simulates thru 22 CALLs and RETs, stops at B_DONE
                       (takes 8 seconds)
13) CE PROG          * stacks FD8CH, jumps to PROG
    IN 2               simulates thru two CALLs, stops at C
    OUT 2              simulates C once, simulates B, executes C nine
                       times, stops at B_DONE (takes 1 second)
14) CE PROG          * stacks FD8CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GE                 simulates PROG, B, and C, stops at B_DONE (takes
                       10 seconds)
15) CE PROG          * stacks FD8CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GO                 simulates PROG, executes B, stops at B_DONE
-------------------------------------------------------------------
16) CE PROG          * stacks FD8CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GE                 simulates five instructions, stops at C
    GE 10              simulates thru nine C break points, stops at EXIT
                       (takes 10 seconds)
17) CE PROG          * stacks FD8CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GO                 simulates PROG, executes B, stops at EXIT (CALL C
                       is executed thus hiding the break point on C)
```

Appendix J: **DBG Simulation Examples** (TANDY 200)

These example command sequences operate on this program. Use TEXT to enter it into a file named PROG.DO. The ORG value can be changed to a different value, if 61000 is not suitable.

```
          ORG 61000

PROG:     LXI D,-2          ;entry to main program
          LXI H,2
          CALL B
B_DONE:   DAD D
EXIT:     RET

B:        MVI B,0           ;entry to subroutine B
LOOP_B:   CALL C            ;loop 10 times on five instructions
          INR B
          MOV A,B
          CPI 10
          JNZ LOOP_B
          RET

C:        MVI C,0           ;entry to subroutine C
LOOP_C:   INR C             ;loop 248 times on four instructions
          MOV A,C
          CPI 248
          JNZ LOOP_C
          RET
```

Enter BASIC and allocate user high memory area.

```
Ok
CLEAR 256,61000 (use your ORG value, if different)
```

Assemble the program.

```
Ok
(function key 6)
CMD> ASMN PROG
Polar Engr (c)1985
0 error(s).
```

Enter DBG.

```
Ok
(function key 6)
CMD> DBG
Polar Engr (c)1985
F88C CALL F4D4    Z NC PE P SP=EB28.2906
 A=04 BC=00BE.50 DE=0000.C3 HL=0850.08
DBG> DBG command
```

Each of the following examples assume that you have just entered DBG.

```
1. CALL PROG          * immediate assembly command executes PROG,
                        returns to F88CH
2. J PROG             * stacks F88CH, jumps to PROG
                        simulates PROG, executes B, returns to F88CH
```

Appendix J: **DBG Simulation Examples** (TANDY 200—continued)

```
 3) CE PROG          * stacks F88CH, jumps to PROG
    N 3                simulates three instructions, stops at B
 4) CE PROG          * stacks F88CH, jumps to PROG
    IN                 simulates three instructions, stops at B
 5) CE PROG          * stacks F88CH, jumps to PROG
    1 B                set break point 1 at B
    GE                 simulates three instructions, stops at B
 6) CE PROG          * stacks F88CH, jumps to PROG
    1 B                set break point 1 at B
    GO                 simulates three instructions, stops at B
                       (DBG simulates the jump part of CALL B and
                       encounters the break point)
-------------------------------------------------------------------
 7) CE PROG          * stacks F88CH, jumps to PROG
    S 3                simulates two instructions, executes B, (three
                       total) stops at B_DONE
 8) CE PROG          * stacks F88CH, jumps to PROG
    SE 3               simulates two instructions, simulates B and C,
                       (three total) stops at B_DONE (takes 10 seconds)
 9) CE PROG          * stacks F88CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    OUT                simulates B, executes C 10 times, stops at B_DONE
10) CE PROG          * stacks F88CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    S 5*10+2           simulates 42 instructions, executes C 10 times,
                       (52 steps) stops at B_DONE
11) CE PROG          * stacks F88CH, jumps to PROG
    N 3                simulates three instructions, stops at B
    N 2+10*(5+2+248*4) simulates 9992 instructions, stops at B_DONE
                       (takes 10 seconds)
12) CE PROG          * stacks F88CH, jumps to PROG
    IN 2+10*2          simulates thru 22 CALLs and RETs, stops at B_DONE
                       (takes 8 seconds)
13) CE PROG          * stacks F88CH, jumps to PROG
    IN 2               simulates thru two CALLs, stops at C
    OUT 2              simulates C once, simulates B, executes C nine
                       times, stops at B_DONE (takes 1 second)
14) CE PROG          * stacks F88CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GE                 simulates PROG, B, and C, stops at B_DONE (takes
                       10 seconds)
15) CE PROG          * stacks F88CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GO                 simulates PROG, executes B, stops at B_DONE
-------------------------------------------------------------------
16) CE PROG          * stacks F88CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GE                 simulates five instructions, stops at C
    GE 10              simulates thru nine C break points, stops at EXIT
                       (takes 10 seconds)
17) CE PROG          * stacks F88CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GO                 simulates PROG, executes B, stops at EXIT (CALL C
                       is executed thus hiding the break point on C)
```

Appendix J: **DBG Simulation Examples** (NEC 8201A)

These example command sequences operate on this program. Use TEXT to enter it into a file named PROG.DO. The ORG value can be changed to a different value, if 62200 is not suitable.

```
          ORG 62200

PROG:     LXI D,-2          ;entry to main program
          LXI H,2
          CALL B
B_DONE:   DAD D
EXIT:     RET

B:        MVI B,0           ;entry to subroutine B
LOOP_B:   CALL C            ;loop 10 times on five instructions
          INR B
          MOV A,B
          CPI 10
          JNZ LOOP_B
          RET

C:        MVI C,0           ;entry to subroutine C
LOOP_C:   INR C             ;loop 248 times on four instructions
          MOV A,C
          CPI 248
          JNZ LOOP_C
          RET
```

Enter BASIC and allocate user high memory area.

```
Ok
CLEAR 256,62200 (use your ORG value, if different)
```

Assemble the program.

```
Ok
(function key 6)
CMD> ASMN PROG
Polar Engr & Cons (c)1984
0 error(s).
```

Enter DBG.

```
Ok
(function key 6)
CMD> DBG
Polar Engr & Cons (c)1984
FC8C CALL F992    Z NC PE P SP=F160.2010
 A=04 BC=00BE.81 DE=0000.C3 HL=0850.08
DBG> DBG command
```

Each of the following examples assume that you have just entered DBG.

```
          PROG          * immediate assembly command executes PROG,
                          returns to FC8CH
          PROG          * stacks FC8CH, jumps to PROG
          RET             simulates PROG, executes B, returns to FC8CH
```

Appendix J: **DBG Simulation Examples** (NEC 8201A--continued)

```
 3) CE PROG          * stacks FC8CH, jumps to PROG
    N 3                simulates three instructions, stops at B
 4) CE PROG          * stacks FC8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
 5) CE PROG          * stacks FC8CH, jumps to PROG
    1 B                set break point 1 at B
    GE                 simulates three instructions, stops at B
 6) CE PROG          * stacks FC8CH, jumps to PROG
    1 B                set break point 1 at B
    GO                 simulates three instructions, stops at B
                       (DBG simulates the jump part of CALL B and
                       encounters the break point)
-------------------------------------------------------------------
 7) CE PROG          * stacks FC8CH, jumps to PROG
    S 3                simulates two instructions, executes B, (three
                       total) stops at B_DONE
 8) CE PROG          * stacks FC8CH, jumps to PROG
    SE 3               simulates two instructions, simulates B and C,
                       (three total) stops at B_DONE (takes 10 seconds)
 9) CE PROG          * stacks FC8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    OUT                simulates B, executes C 10 times, stops at B_DONE
10) CE PROG          * stacks FC8CH, jumps to PROG
    IN                 simulates three instructions, stops at B
    S 5*10+2           simulates 42 instructions, executes C 10 times,
                       (52 steps) stops at B_DONE
11) CE PROG          * stacks FC8CH, jumps to PROG
    N 3                simulates three instructions, stops at B
    N 2+10*(5+2+248*4) simulates 9992 instructions, stops at B_DONE
                       (takes 10 seconds)
12) CE PROG          * stacks FC8CH, jumps to PROG
    IN 2+10*2          simulates thru 22 CALLs and RETs, stops at B_DONE
                       (takes 8 seconds)
13) CE PROG          * stacks FC8CH, jumps to PROG
    IN 2               simulates thru two CALLs, stops at C
    OUT 2              simulates C once, simulates B, executes C nine
                       times, stops at B_DONE (takes 1 second)
14) CE PROG          * stacks FC8CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GE                 simulates PROG, B, and C, stops at B_DONE (takes
                       10 seconds)
15) CE PROG          * stacks FC8CH, jumps to PROG
    1 B_DONE           set break point 1 at B_DONE
    GO                 simulates PROG, executes B, stops at B_DONE
-------------------------------------------------------------------
16) CE PROG          * stacks FC8CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GE                 simulates five instructions, stops at C
    GE 10              simulates thru nine C break points, stops at EXIT
                       (takes 10 seconds)
17) CE PROG          * stacks FC8CH, jumps to PROG
    1 C                set break point 1 at C
    2 EXIT             set break point 2 at EXIT
    GO                 simulates PROG, executes B, stops at EXIT (CALL C
                       is executed thus hiding the break point on C)
```

INDEX

Topic Index

ROM Routine Index