HEWLETT-PACKARD

Assembler ROM Manual

HP-83/85

HEWLETT PACKARD

HP-83/85

ASSEMBLER ROM

AND

HP-82928A SYSTEM MONITOR

MANUAL

00085-90444 Rev. B

November I981

Printed in U.S.A.

CONTENTS

NOTES

INTRODUCTION

This manual outlines the commands, statements, instructions and use of both the HP-83/85 Assembler ROM and the HP-82928A System Monitor. The manual is not tutorial in nature and it assumes that you already have at least some knowledge of programming in assembly language. If you are not already familiar with the HP-83 or HP-85 Personal Computer, you should read the owner's manual before proceeding.

The HP-83/85 contains both read-only memory (ROM) and read-write or random-access memory (RAM). The RAM contains the user's BASIC language programs and data, and can also contain a binary (machine language) program. The ROM contains the machine language program which recognizes and executes the statements provided by the BASIC language. Thus, the operating system ROM in the HP-83/85 provides such statements as PRINT, DISP, and INPUT.

When external peripheral devices are added, their wider range of capabilities requires more extensive BASIC language statements to fully use these capabilities. Additional plug-in modules, called add-on ROMs, merely enrich the BASIC language by increasing the number of statements and functions that can be recognized and executed. Similarly, a binary program within the computer also extends the BASIC language.

THE ASSEMBLER ROM

Using the Assembler ROM, you can write assembly-language binary programs for residence and execution within the computer or for creation of a plug-in EPROM for the computer. A binary program can:

Extend the BASIC language:

--Provide new BASIC statements and system functions.
--Take over and redefine existing BASIC statements and functions.
--Expand I/O control.

Give increased execution speed:

--Yield faster results.
--Speed up I/O processes.

Redefine the system:

--Take over system "hooks," giving access to the HP-83/85 operating system.
--Implement languages other than BASIC.
--Redefine the use and operation of I/O.

A ROM program is written in virtually the same manner as a binary program--the main difference is in how the program is used after assembly--and in this manual both are often termed simply "binary programs."

When connected to an HP-83/85 Personal Computer, the Assembler ROM permits you to enter and edit source code for binary programs right on the computer's CRT screen. Automatic line numbering and cursor movement are active, and the source code can be stored on a mass storage device such as a tape cartridge or disc, listed, and edited in much the same way a BASIC program is stored, listed, and edited. As source statements are entered, they are automatically checked for syntax errors and duplicate labels.

At assembly time, the resulting object code (machine language) is stored on a mass storage device such as a tape or disc. This object code can also be loaded automatically or on command into the HP-83/85, and it is then ready to run.

To aid in programming, a tape cartridge and a disc are provided with the Assembler ROM. Each of these contains a global file of HP-83/85 system labels and their memory addresses for use during assembly. The tape and disc also contain useful sample programs to help illustrate how binary programs are created.

The Assembler ROM gives you the ability to "tailor" statements for your own applications, to speed up program execution, to perform sophisticated graphics. But with all the power and system accessibility provided by the Assembler ROM, it is also possible to defeat the computer's internal safeguards and even seriously damage the HP-83 or HP-85. For this reason, you should understand assembly language programming before attempting to use the Assembler ROM.

THE HP-82928A SYSTEM MONITOR

The System Monitor is an optional plug-in module that is designed for use only in conjunction with the Assembler ROM. The System Monitor is not required, but it makes the debugging and modification of binary programs much easier.

With the System Monitor module attached, you can set breakpoints that interrupt the execution of a program. After program execution has been interrupted, you can examine or change the contents of memory, you can execute one instruction at a time (single-step), or you can trace the operation of a machine language program, printing the status of the CPU after each instruction.

SCOPE OF THIS MANUAL

This manual contains information about three separate products:
--The HP-83/85 Personal Computer and its operating system.
--The Assembler ROM.
--The HP-82928A System Monitor.

The manual has been written to help you most effectively use these three products together. If you are looking for information in a specific area, however, you may want to refer to the manual sections as outlined below:

THE COMPUTER'S OPERATING SYSTEM

Manual Section	Topic
3	CPU Structure and Operation
5	System Architecture and Operation
7	HP-83/85 System Routines
Appendix A	Glossary of Terms
Appendix B	Hardware Diagram
Appendix E	ASCII Table
Appendix F	Tables of Tokens and Attributes

THE ASSEMBLER ROM

Manual Section	Topic
Introduction	
1	Getting Started
2	Assembler Commands and Statements
4	HP-83/85 Assembler Instructions
6	Writing Binary and ROM Programs
8	Sample Binary Programs
Appendix C	Assembler Instructions
Appendix D	Decoding Assembler Instructions
Appendix G	Error Messages

THE SYSTEM MONITOR

Manual Section	Topic
1	Getting Started
9	The HP-82928A System Monitor

SYNTAX GUIDELINES

The syntax used in this manual for illustrating commands, statements, and instructions is shown here:

LDB — Instructions shown in capital letters, but not underlined, must be entered exactly as shown (in either upper-case or lower-case letters).

<u>DR</u> — Items shown underlined are expressions or names that must be specified in the instruction, statement, or command.

[ ] — Items shown between brackets are optional. If several items are stacked between brackets, any one or none of the items may be specified.

... — Three dots (ellipsis) following a set of brackets indicate that the items between the brackets may be repeated.

All values for registers and addresses in this manual are octal values. Other values (numbers, quantities, etc.) are given in decimal base unless otherwise noted.

NOTES

SECTION 1

GETTING STARTED

When shipped from the factory, the HP-83/85 Assembler ROM package comprises the following items:

--HP-83/85 Assembler ROM, part number 00085-15007.

--HP-85 Assembler Global File tape.

--HP-83 Assembler Global File disc.

--HP-83/85 Assembler ROM Manual, part number 00085-90444.

--HP-83/85 Assembler ROM Pocket Guide, part number 00085-90445.

To use the Assembler ROM, you will need at least the following:

--HP 82936A ROM Drawer

AND

--HP-83 Personal Computer with Flexible Disc Drive

OR

--HP-85 Personal Computer with or without Disc Drive attached.

In addition, to help you write and de-bug binary programs with the Assembler ROM, you may also wish to obtain the HP-82928A System Monitor.

This manual gives installation and operation instructions for the HP-83/85 Assembler ROM and its global file, and also for the HP-82928A System Monitor.

ROM INSTALLATION

Install the HP-83/85 Assembler ROM in one of the six slots in an HP 82936A ROM Drawer. The ROM drawer can then be plugged into one of the four module ports in the rear of the computer. If you are unfamiliar with the procedure for installing a ROM and a ROM drawer, refer to the owner's manual for your computer, or to the HP 82936A ROM Drawer Instruction Sheet for the proper procedure.

TAPE CARTRIDGE OR DISC INSTALLATION AND USE

To install the tape cartridge containing the global file and the example binary programs into the HP-85 computer, follow the instructions in the HP-85 Owner's Manual.

To install the disc containing the global file and sample binary programs, follow the instructions in the owner's manual for the Flexible Disc Drive.

As part of the process of assembling a binary program, the object code is stored on a mass storage device such as a tape or disc. If, as will probably be most convenient, you wish to use the global file tape cartridge for this purpose, make sure that the tab on the cartridge is set to RECORD.

Here is a list of the files available on the global file tape and disc. Files with names ending in "S" are source code files. Files with names ending in "B" are binary program object code files. (The file GLOBAL is an ASCII data file containing the assembled global file.)

File	Description
FTOCS, FTOCB	Example program: Fahrenheit to Celsius.
GCURS, GCURB	Example program: Implements a graphics cursor.
SOFTKS, SOFTKB	Example program: Special function keys as typing aids.
UDLS, UDLB	Example program: Underlines a string.
RECPLS, RECPLB	Example program: Rectangular/polar conversions.
ROMPRS, ROMPRB	Example program: Rectangular/polar conversions. (Written for a ROM.)
GL01S, GL02S	Global file in source code. (Two parts.)
GLOBAL	Global file.

SYSTEM MONITOR INSTALLATION

The HP-82928A System Monitor is installed in one of the four module I/O ports of the HP-83 or HP-85. To install the System Monitor, follow the instructions in the owner's manual for your computer.

The System Monitor is not required for use of the Assembler ROM.

ASSEMBLER ERRORS

The Assembler ROM and the System Monitor contain some error messages of their own. A complete list of these error messages and their causes may be found in appendix G of this manual.

Because of the ability of binary programs to take over internal HP-83/85 routines and to defeat safeguards within the computer, it is possible to physically damage the computer without halting execution or even generating an error. For example, a flawed binary program could hold the print head element on and burn it out, or it could run the magnetic tape in an HP-85 tape cartridge off the end of the spool. For this reason, you should be extremely careful as you write and run binary programs, particularly if your programs take over any of the internal printer or tape routines.

CAUTION

> If during the running of a binary program the print head appears to be "locked up" or an HP-85 tape cartridge begins to unspool, shut off the computer's power switch immediately.

NOTES

SECTION 2

ASSEMBLER COMMANDS, STATEMENTS, AND FUNCTIONS

When the Assembler ROM is attached to the HP-83 or HP-85, it provides:

--Assembler commands

--Assembler statements and functions

--Assembly language elements

The commands and the statements and functions provided by the Assembler ROM are added to the functions, statements and commands that are already part of the computer's instruction set. They are executed exactly as the rest of the computer's instruction set, and have been created to help the programmer control and use the assembler.

Assembly language elements are used as the actual instructions in writing binary programs. The format and use of these elements are discussed in section 4 of this manual, and a complete list of them may be found in that section and in appendix C.

ASSEMBLER COMMANDS

A command is non-programmable, and can be executed only from the keyboard (i.e., in calculator mode). The assembler commands permit the user to transfer between assembler and BASIC system modes, to assemble, store and load binary program source code, and to find labels within the source code in memory.

Assembler commands may be entered as normal calculator mode statements, alone on a line and terminated by [END LINE]. In addition, in assembler mode, the computer's special function keys and certain other keys will generate the assembler commands as follows:

Key	Assembler Command
[LOAD]	ALOAD
[RUN]	ASSEMBLE
[STORE]	ASTORE
[K1]	BASIC
[K2]	FLABEL
[K3]	FREFS

ALOAD

Assembler Command

Load Source Code

Format: ALOAD "file name"

Description: Legal only in assembler mode. Loads source code that was previously stored with the ASTORE command into the computer's memory from the file specified on the currently-selected mass-storage device. The file must be of the type known as "extended" (****). In assembler mode, the [LOAD] key is a typing aid for the word ALOAD.

Example: ALOAD "OXY"

NOTE

The "extended" type of file, denoted by **** on the directory of a mass storage device, does not necessarily mean that the file contains source code. In fact, other HP-83/85 firmware and software may generate extended type files.

ASSEMBLE

Assembler Command

Assemble Source Code

Format: ASSEMBLE "file name" [, numeric value]

Description: Legal only in assembler mode. Assembles source code currently in the computer's memory and stores it in the file specified by file name on the currently selected mass storage device (e.g., tape or disc). The assembled source code is stored as either a binary program or, if the file has been declared a ROM or global file, as a series of strings in a data file.

If at assembly numeric value is evaluated as zero, the binary program currently in the computer's memory is scratched, and the object code of the newly-assembled binary program is loaded from the mass storage device into memory. Default numeric value is evaluated as zero.

If at assembly numeric value is other than zero, any binary program currently in memory remains inviolate, and the object code of the newly-assembled binary program is stored only on the current mass storage device.

In assembler mode, the [RUN] key is a typing aid for the word ASSEMBLE.

CAUTION

If a program contains an error or if programs are linked at assembly, this command can destroy the source code; if the source code is to be saved on a mass storage device such as a disc or tape cartridge, it should be stored there before typing ASSEMBLE.

Examples: ASSEMBLE "CENT" Assembles source code into object code, stores object code as a file named CENT on the tape cartridge or disc, and performs a LOADBIN "CENT" to load the object code.

ASSEMBLE "OXY", 3 Assembles source code into object code and stores object code as a file named OXY on the tape cartridge or disc.

ASSEMBLER

Assembler Command

Switch to Assembler Mode

Description: Legal only when the computer is in normal system mode, this command scratches memory and puts the computer into assembler mode. In assembler mode, most normal BASIC statements will still operate, but only as calculator mode statements--they are not programmable. Source code for a binary program can then be typed in with line numbers, just as a BASIC program is typed in while in normal system mode (but with only one instruction per line). Unlike its operation in normal system mode, the computer is somewhat sensitive to character spacing while in assembler mode. Auto line numbering, screen editing, listing, etc., are all functional. The [CONT], [STEP], and [INIT] keys are inoperative in assembler mode; in this mode the [RUN] key acts as a typing aid for the word ASSEMBLE.

Displays the word Ready when executed.

ASTORE

Assembler Command

Store Source Code

Format: ASTORE "file name"

Legal only in assembler mode. Stores the source code currently in the computer's memory into the specified file on the currently-selected mass storage device (e.g., tape or disc). File is of the type known as "extended," shown in the directory as ****.

In assembler mode, the [STORE] key is a typing aid for the word ASTORE.

Example: ASTORE "OXY"

BASIC

Assembler Command

Switch to BASIC Mode.

Format: BASIC

Description: Legal only when in assembler mode, this command scratches memory and puts the HP-83/85 back into normal BASIC mode.

Displays the word Ready when executed.

In assembler mode, special function key [K1] acts as a typing aid for the word BASIC.

FLABEL
Assembler Command

Find Label

Format: FLABEL "label"

Description: Legal only in assembler mode. This command searches through the source code in memory for the label specified. For each occurrence of the label (as a label at the beginning of a line) the line is listed. After an FLABEL command has been executed, pressing the [LIST] key causes the source code to be listed, beginning with the last line where the label occurs.

In assembler mode, special function key [K2] may be used as a typing aid for the word FLABEL.

Examples: FLABEL "SIN"

FLABEL "PARSIT"

FREFS
Assembler Command

Find References to Labels

Format: FREFS "label"

Description: Legal only in assembler mode. Searches through the source code in memory for all occurrences, whether at the beginning of a line or not, of the specified label. Otherwise operates the same as FLABEL, including the operation of the [LIST] key.

In assembler mode, special function key [K3] acts as a typing aid for the word FREFS.

Examples: FREFS "SIN"

FREFS "CENT"

TREM

Assembler Command

Toggle Remarks

Format: TREM

Description: Legal only in assembler mode. Toggles an internal flag to suppress end-of-line comments and prevent them from appearing on the computer's CRT when source code is listed. Default condition is that end-of-line comments are not shown on the CRT. Because end-of-line comments can wrap around on the CRT, this command can make the CRT display of source code more easily readable.

ASSEMBLER STATEMENTS AND FUNCTIONS

Statements and functions are programmable BASIC language elements. The statements and functions provided by the Assembler ROM are simply additions to the BASIC language of the HP-83/85 computer. As with all BASIC statements and functions, they may be used either in calculator mode or as part of a BASIC program when the HP-83/85 is in normal BASIC system mode. When the computer is in assembler mode, of course, all BASIC statements and functions may be executed only from the keyboard (i.e., as calculator mode statements).

DEC

Assembler-Provided BASIC Function

Octal to Decimal

Format: DEC (<u>octal numeric value</u>)

Description: Returns the decimal equivalent of the specified octal value.

Example: DEC (377) Returns 255, the decimal equivalent of 377_8.

MEM
Assembler-Provided BASIC Statement

Memory Dump

Format: MEM address [:ROM #] [,# of bytes] [=#,#, ...]

Description: Dumps the contents of computer RAM or ROM memory to the current CRT IS device beginning with the octal address. Continues dumping for the specified octal [,# of bytes]. At power-on, default # of bytes is 100_8; otherwise, default is the last # of bytes specified.

The [:ROM #], if included, is a decimal value that selects the plug-in ROM from which memory is dumped. At power-on, default value for ROM # is 0; otherwise, default is the last ROM # specified.

The output is in two forms: The first shows the octal representation of the bytes in memory; the second shows the ASCII representation of the bytes.

If =#,# is included in the statement, memory is not dumped, but instead the contents of memory locations beginning at address are changed to the octal values specified after the = sign. The memory locations must be in RAM (32K-64K). The contents of one succeeding memory location are changed for each value specified after the = sign. The # of bytes, if included in the statement, is disregarded in this case.

Examples: MEM 103300 Dumps contents of 100_8 bytes of memory to the CRT IS device, beginning with memory location 103300.

MEM 103300, 20 Dumps contents of 20_8 bytes of memory to the CRT IS device, beginning with memory location 103300.

MEM 60200: 40,200 Dumps contents of 200 bytes of Assembler ROM (ROM # 40) to CRT IS device, beginning with memory location 60200.

MEM 105000 = 0,0,0,15 Loads memory locations 105000, 105001, and 105002 with zeros, and loads location 105003 with 15_8.

MEMD

Memory Dump

Assembler-Provided BASIC Statement

Format: MEMD address [: rom#] [,# of bytes] [=#,#,...]

Description: Same as MEM statement, except it reads the contents of two bytes of memory beginning with address and uses those contents as the actual address at which to begin the dump.

Example: MEMD 101233 Dumps contents of 100 bytes of memory to current CRT IS device beginning with location pointed to by value in bytes 101233 and 101234. (Since address 101233 is the address of BINTAB, this statement actually dumps the first 100 bytes of a binary program, if one is resident.)

REL

Assembler-Provided BASIC Statement

Relative Address

Format: REL (octal address)

Description: Returns the absolute address of a relative address. Takes the relative octal address and adds to it the address (called BINTAB) of the beginning of the binary program to yield the octal absolute address. May be used alone or with MEM. May also be used with command BKP if HP-82928A System Monitor is attached.

Examples: REL (0) Returns address of the beginning of the binary program (i.e., the contents of BINTAB).

MEM REL (123), 100 Dumps contents of 100_8 bytes of memory to the CRT IS device, beginning with the 123rd byte of the binary program.

BKP REL (675) Sets break point at byte 675 after the beginning of the binary program. (BKP is available only with the HP-82928A System Monitor attached.)

SCRATCHBIN

Assembler-Provided BASIC Statement

Scratch Binary Program

Format: SCRATCHBIN

Description: Scratches the current binary program from computer memory, without affecting anything else. Nothing can follow SCRATCHBIN on a line except [END LINE].

OCT

Assembler-Provided BASIC Statement

Decimal to Octal

Format: OCT (<u>decimal numeric value</u>)

Description: Returns the octal equivalent of the specified decimal value.

Example: OCT (45) Returns 55, the octal equivalent of 45_{10}.

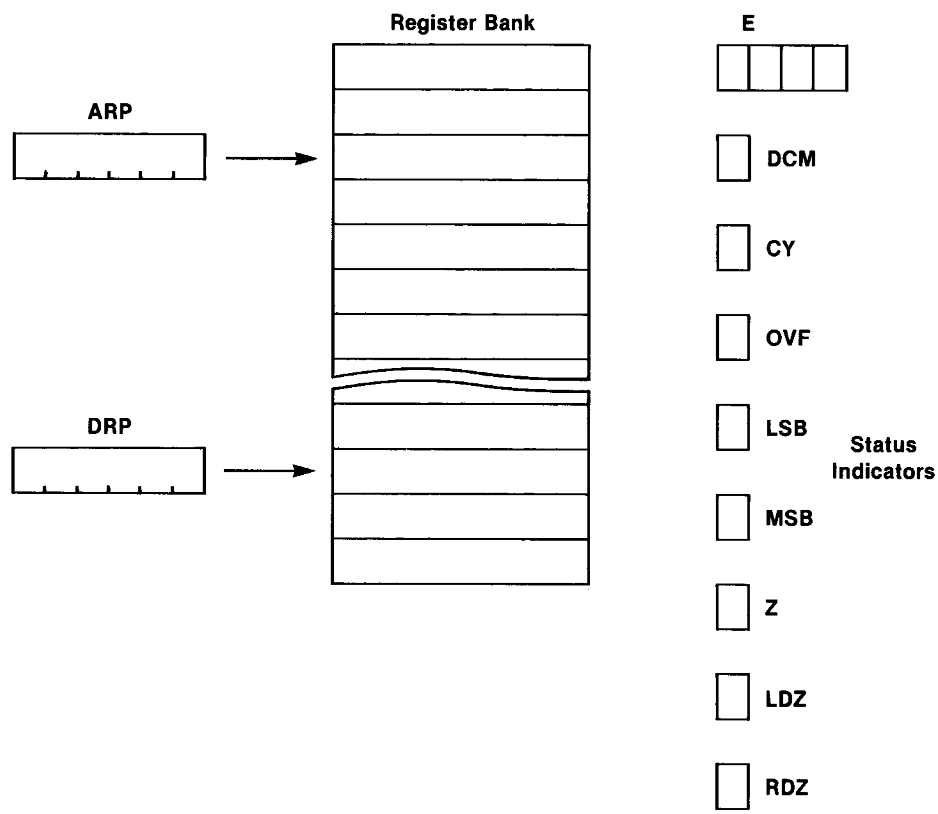
NOTES

SECTION 3

CPU STRUCTURE AND OPERATION

This section explains the structure, addressing modes and operation of the central processing unit (CPU) in the HP-83/85.

The HP-83/85 CPU consists of a 64_{10}-byte register bank, a pair of address pointers called the address register pointer (ARP) and the data register pointer (DRP), an arithmetic and logic unit (ALU) and a shifter, and a set of status indicators.



CENTRAL PROCESSING UNIT

ARP AND DRP

The address register pointer (ARP) and the data register pointer (DRP) are independent six-bit CPU locations. Both the ARP and the DRP can be used to address any of the bytes in the CPU register bank.

The CPU register addressed by the ARP is called the address register, or AR. The register addressed by the DRP is called the data register, or DR.

CPU REGISTER BANK

The heart of the CPU is the register bank of 64 8-bit bytes of random-access memory. These bytes form registers which are grouped into two-byte (16-bit) sections and eight-byte (64-bit) sections. The diagram on the following page shows the organization of the CPU registers, which are numbered from 0 to 77_8, and specified by RØ - R77.

Some of the registers in the CPU register bank are dedicated by hardware to specific tasks.

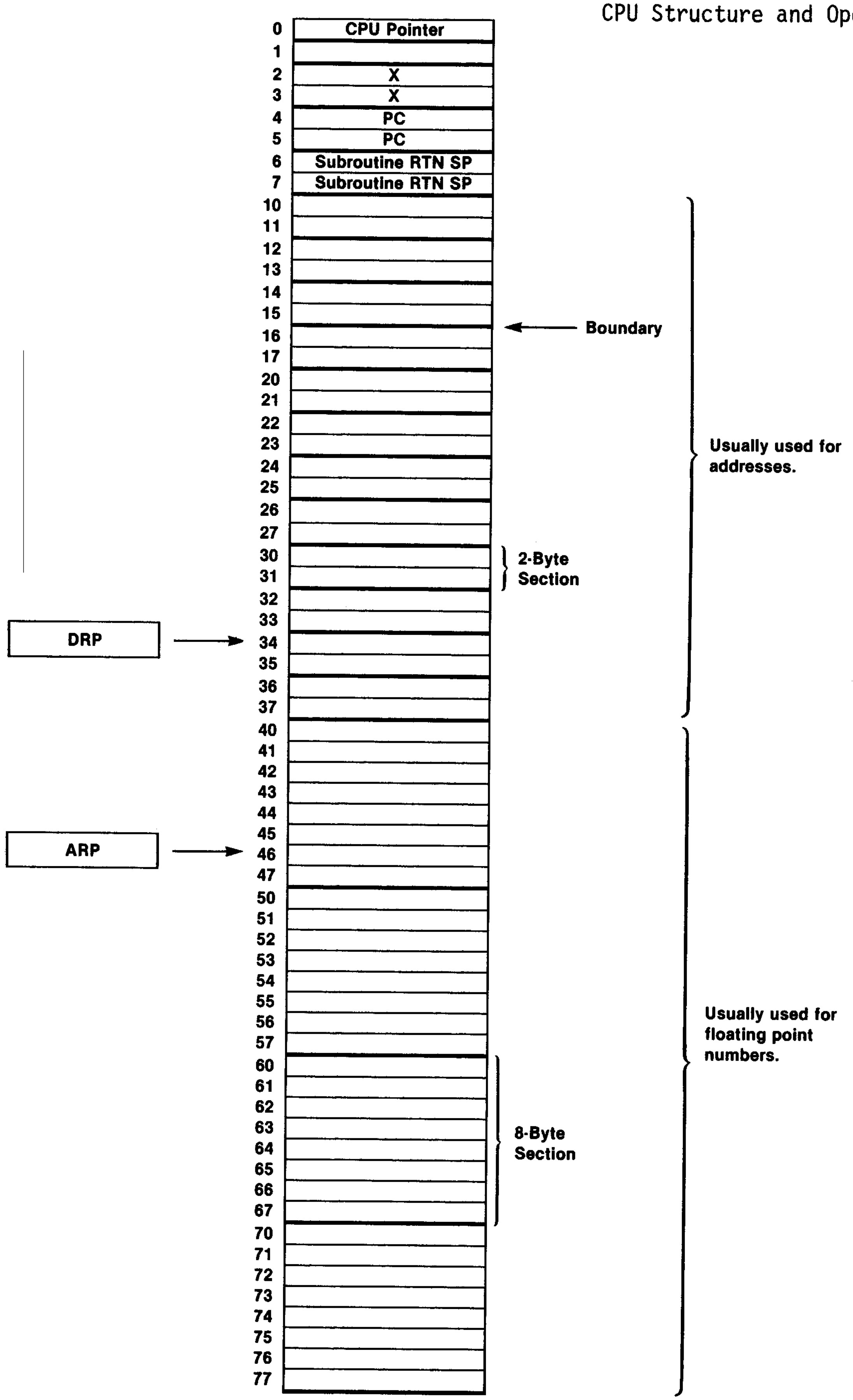
HARDWARE-DEDICATED REGISTERS

The first 40_8 registers of the CPU (RØ - R37) are divided into two-byte (16-bit) sections. Of these, many of the bytes are reserved by hardware for use as special-purpose registers. These hardware-dedicated registers are:

Register Bank Pointer. Register 0 is a pointer to the remainder of the CPU register bank. Register 1 is inaccessible except through register 0.

Index Scratch. Registers 2 and 3 are scratch registers used for indexed addressing (X). Their contents are destroyed by execution of instructions using indexed addressing.

Program Counter. Registers 4 and 5 contain the program counter (PC).



CPU REGISTER BANK

Return Stack Pointer. Registers 6 and 7 contain the pointer for the subroutine return stack. (The space allocated for this stack in the computer's system memory comprises addresses 101300 through 101777, although sometimes these addresses may be used for other purposes.)

In addition to the special-purpose registers described above, certain other CPU registers are commonly used for specific purposes by internal HP-83/85 routines. (For example, registers R40 and R50 are used by internal mathematics routines for addition, subtraction, etc.)

REGISTER BOUNDARIES

The CPU registers are separated by boundaries, shown as heavy lines in the illustration of the register bank above. In the first 32 bytes, there is a boundary every two bytes. In the next 32 bytes, there is a boundary every eight bytes.

This partitions the first 32 bytes into 16-bit sections (used primarily for address manipulation) and the next 32 bytes into 64-bit sections (used primarily for floating point quantities). The register array is, therefore, capable of holding up to four floating-point numbers and twelve 16-bit addresses.

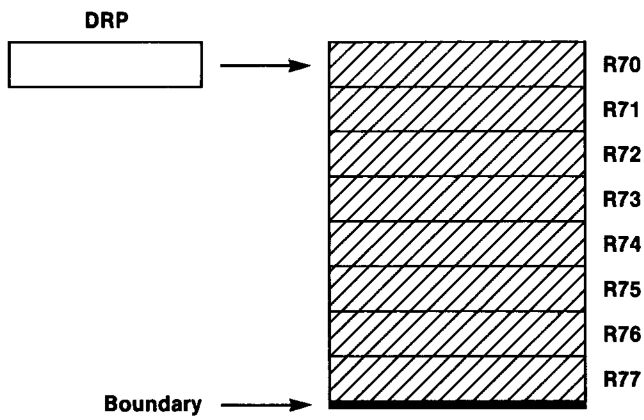
MULTI-BYTE OPERATIONS

The HP-83/85 CPU structure permits "multi-byte operations," involving a string of bytes rather than just a single byte. A string can consist of from one to eight consecutive CPU registers. The exact number is determined by the DRP and the next boundary.

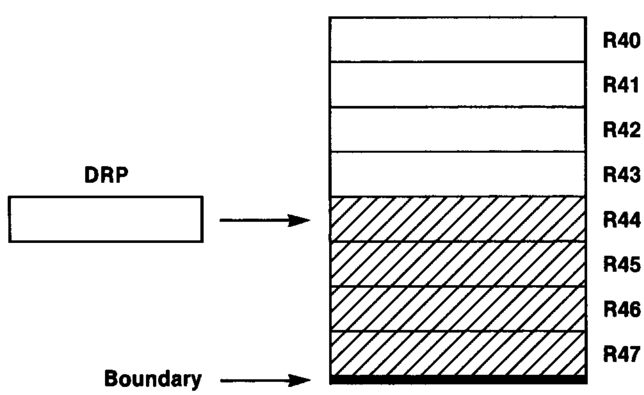
The locations involved in a multi-byte operation are those beginning with the location pointed to by the DRP and ending with the next boundary. The next boundary is the one in the direction of increasing addresses (except in the case of a shift right instruction.)

The following examples should help explain this concept:

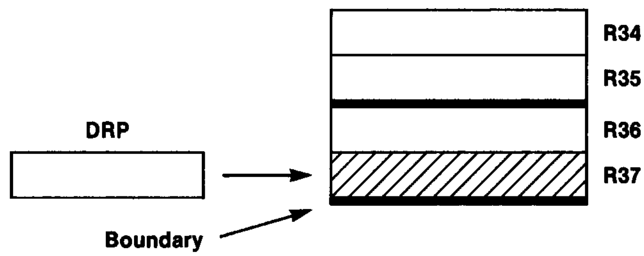
--A multi-byte increment with DRP set to 70 (that is, executing ICM R70) results in an increment of the 64-bit quantity stored between locations R70 and R77. Higher addresses always refer to more significant bytes.



--A multi-byte test with DRP set to 44 (that is, executing TSM R44) results in the status flags being set according to the data found in registers R44, R45, R46 and R47. Location R47 is the most significant byte.

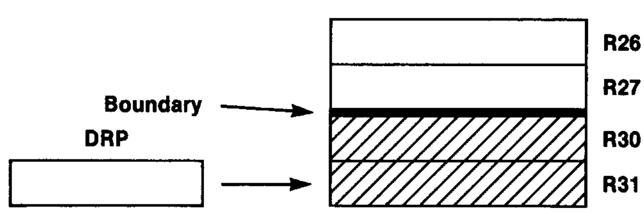


--A multi-byte complement with DRP set to 37 (that is, executing TCM R37) complements only R37.



The only exception to the rule that the next boundary is in the direction of increasing addresses is the shift right instruction. If a multi-byte instruction is a shift right, then the next boundary is the one in the direction of decreasing addresses.

Thus:
--A multi-byte shift right with DRP set to 31 (that is, executing LRM R31) shifts the combined contents of R31 and R30 right. R31 is the most significant byte.



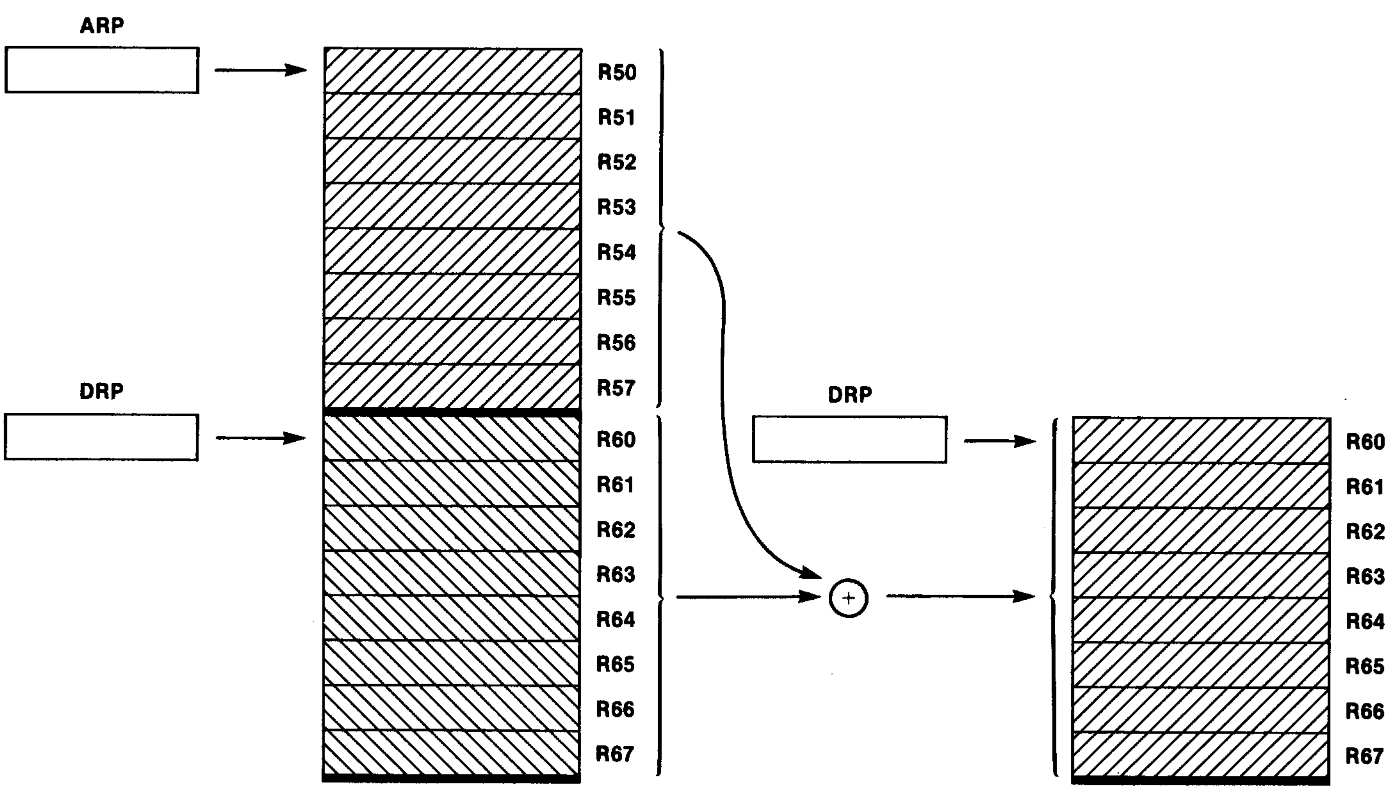
SINGLE-BYTE OPERATIONS

Besides executing multi-byte instructions, the HP-83/85 CPU also executes instructions using single bytes. In a single-byte operation, the DRP refers to only a single byte.

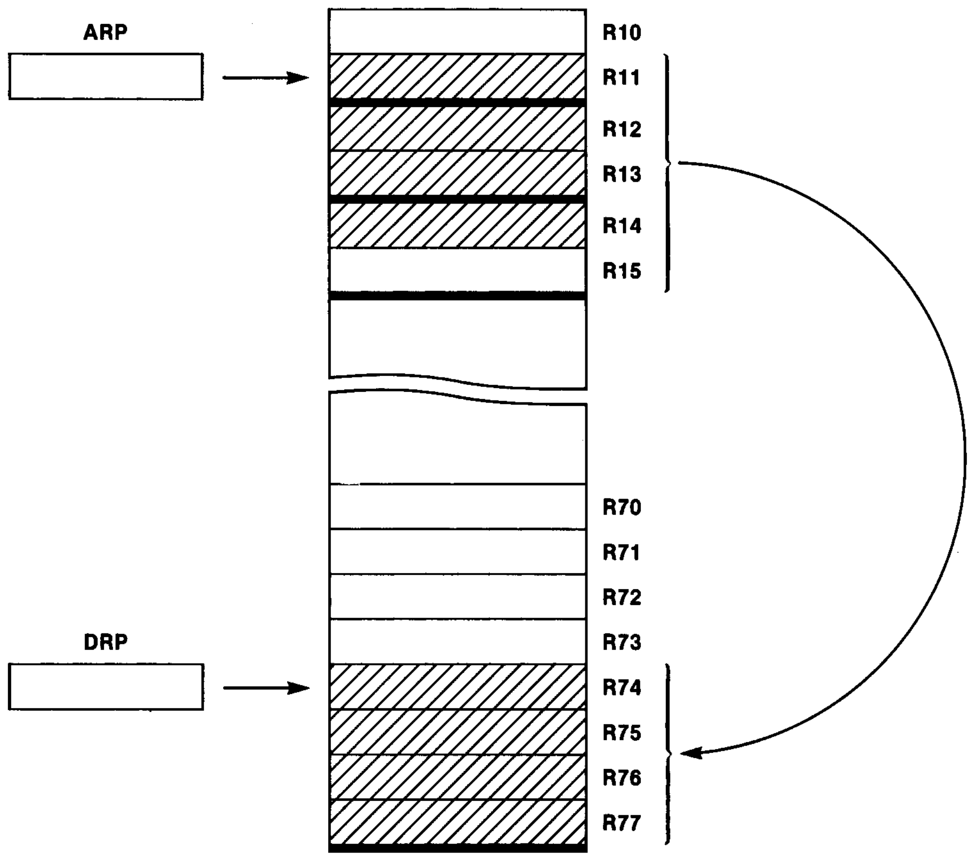
TWO-OPERAND OPERATIONS

Two-operand multi- and single-byte instructions may also be executed. In the case of a multi-byte two-operand instruction, DRP points to the first operand and ARP points to the second. DRP is still used to determine the number of bytes involved for the first operand. The other operand consists of the same number of bytes, beginning with the location to which the ARP points. For example:

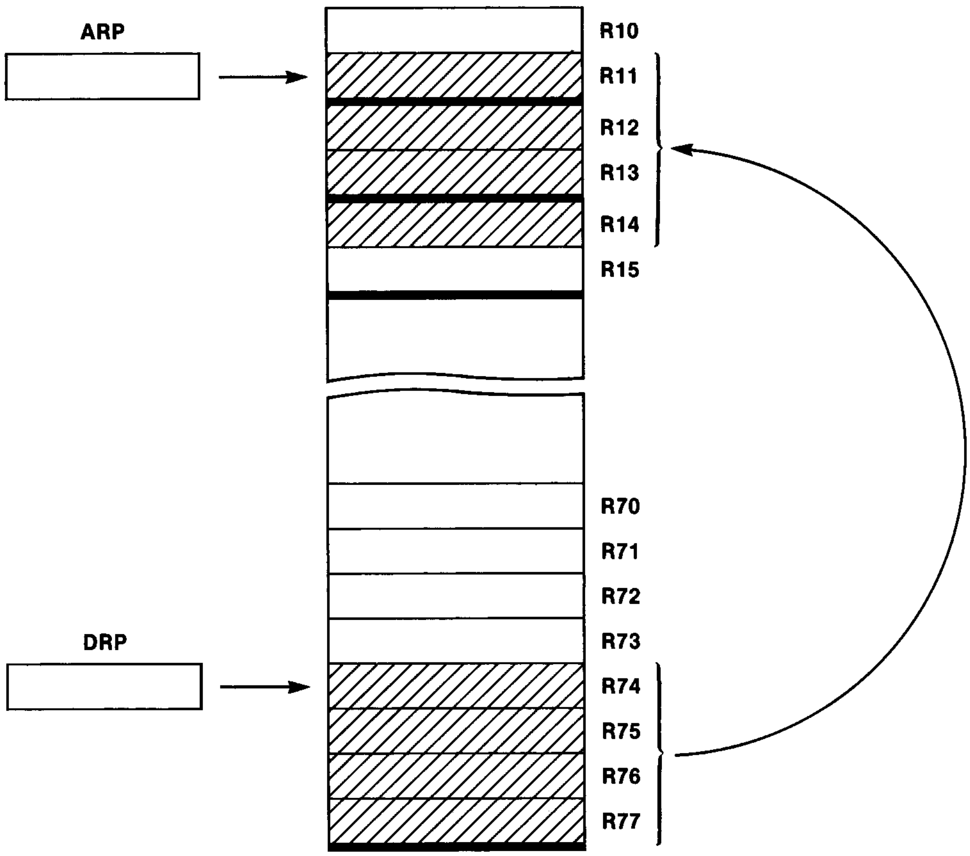
--A multi-byte add with DRP set to 60 and ARP set to 50 (that is, executing ADM R60, R50) results in the 64-bit quantity starting with R50 being added to the 64-bit quantity starting with R60. The sum is stored in R60 through R67.



--A multi-byte load with DRP set to 74 and ARP set to 11 (that is, executing LDM R74, R11) transfers the contents of four bytes beginning with R11 to locations R74, R75, R76 and R77.



--A multi-byte store with DRP set to 74 and ARP set to 11 transfers the contents of R74 through R77 to the four consecutive locations beginning with R11.



Remember: The number of bytes in a multi-byte operation is always determined by the setting of DRP (not ARP) and the next boundary.

There are also two-operand operations where the DRP points to one operand and the second is located in the computer's memory. Once again, the number of bytes to be operated upon is determined by the DRP. The corresponding number of bytes are accessed from memory beginning with the calculated effective address.

NUMBER REPRESENTATION

Numbers in the HP-83/85 are manipulated in a variety of formats. The user has the option of specifying quantities as octal, BCD or decimal. In addition, the internal quantities used in the HP-83/85 occur in various formats, depending on their use.

ADDRESSES

An address, whether in the CPU register bank or in system memory, is always an octal value that occupies two bytes, or 16 bits. The lower-numbered byte contains the less significant byte of the address, and the higher-numbered byte contains the more significant byte of the address. Only the first byte of the two-byte address is referenced by other instructions.

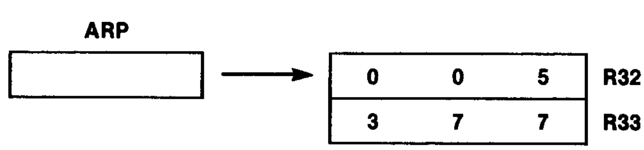
For example, address 177405, translated into a binary quantity, appears like this:

1	7	7	4	0	5	Octal Representation
1	111	111	100	000	101	Binary Representation

When this binary quantity is split into two eight-bit registers, it appears as:

11	111	111	00	000	101	Binary Quantity
3	7	7	0	0	5	Register Contents

Only the first byte of the two-byte address is referenced by other instructions, so an address pointing to ROM location 177405 from the CPU might look like this:



NUMERIC QUANTITIES

Numeric quantities in the HP-83/85 may be of three types: Real, short, and integer. The following illustration shows how numeric quantities are represented internally in the computer. For the illustration, the numbers are shown in CPU registers R40 - R47.

Real

40	E1	E2
41	E0	MS
42	M10	M11
43	M8	M9
44	M6	M7
45	M4	M5
46	M2	M3
47	M0	M1

Integer

45	D1	D0
46	D3	D2
47	S	D4

Short

44	E0	E1
45	M3	M4
46	M1	M2
47	0 0 SM SE	M0

FORMATS OF NUMERIC QUANTITIES

In real or floating-point format, the mantissa is a 12-digit quantity expressed as a magnitude. Each digit consists of four bits. The least significant digit, represented by M11, is stored in R42. The most significant digit, represented by MØ, is stored in R47. The number is normalized; thus, there is an implied decimal point between MØ and M1 in R47. The sign of the mantissa is stored in the least significant digit of R41. A zero is stored as the sign of the mantissa if the number is positive; otherwise, a nine is stored. The exponent is a three-digit number stored in R40 and in the most significant digit position of R41. Exponents are expressed in ten's complement form.

Integer variables are stored in three bytes, with five digits and a sign. Short variables are stored as a mantissa sign (SM) an exponent sign (SE), five mantissa digits, and a two-digit exponent.

STATUS INDICATORS

The HP-83/85 CPU contains eight flags and a four-bit register for program status. The flags signal the present condition of the data, while the four-bit register serves as an "extended" register for counting and data manipulation.

Status can affect or be affected by CPU instructions. In the HP-83/85 CPU, the instruction set has data movement instructions of both the arithmetic and non-arithmetic types. These instructions include:

--Arithmetic: Add, subtract, compare, increment, decrement, complement.

--Non-arithmetic: Load, store, logical and, or, exclusive or, shift, clear, test.

The following status indicators are present in the HP-85 CPU:

E: Extend Register. A four-bit register which can be cleared, incremented, or decremented independent of DCM. Shifts can be made into and out of the extend register only when DCM is set.

DCM: Decimal Mode Flag. When set, binary-coded decimal (BCD) operations will be performed. When cleared, binary operations will be performed. The operations affected by DCM are all the arithmetic data movement instructions and the shift instructions. The DCM flag can be modified only by two CPU instructions, BCD and BIN. The BCD instruction sets DCM, while the BIN instruction clears DCM.

CY: Carry Flag. This one-bit register can be shifted into and out of when DCM is cleared (i.e., BIN mode). It is loaded with the carry from the most significant bit (MSB) according to the table shown here:

CPU Instruction	Carry Flag
Add	CY set according to carry of add.
Subtract	CY set if result is positive, cleared if result is negative.
Compare	Same setting as for subtract.
Increment	CY set as for add.
Decrement	CY set as for subtract.
Shift	CY loaded with bit shifted out, if in binary mode. (Right shift loads CY from LSB.)
Complement	CY cleared by nine's complement, set by ten's complement, if contents of data register (DR) were zero.

All other data movement instructions clear CY.

OVF: Overflow Flag. The overflow flag is set whenever the result of a binary arithmetic operation exceeds the maximum positive or negative number that can be contained in the destination register. This can occur as the result of a compare, binary add, binary subtract, binary complement, or binary left shift instruction. Thus, an arithmetic data movement instruction or a left shift with DCM cleared affects OVF; all other data movement instructions clear OVF. The remaining instructions do not affect OVF.

LSB: Least Significant Bit Flag. LSB is set the same as the least significant bit (LSB) of the result of each data movement instruction.

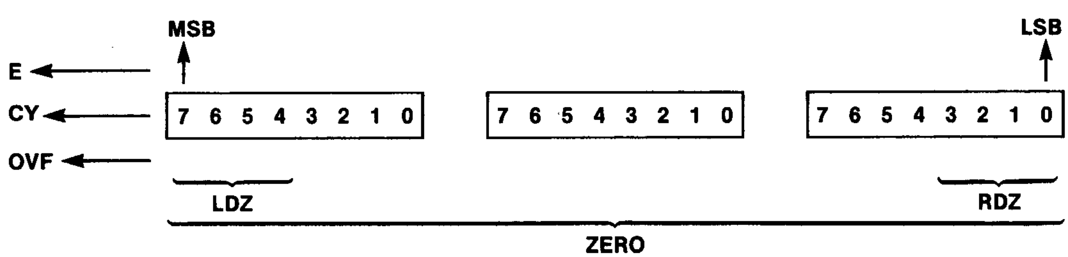
MSB: Most Significant Bit Flag. MSB is set the same as the most significant bit (MSB) of the result of each data movement instruction.

Z: Zero Flag. Z is set if a data movement instruction produces a result of all zeros. If the result is not all zeros, Z is cleared. Other instructions do not affect Z.

LDZ: Left Digit Zero Flag. LDZ is affected only by data movement instructions. LDZ is set if the most significant nibble (four bits) of the result is 0000. If the most significant four bits are not 0000, LDZ is cleared.

RDZ: Right Digit Zero Flag. RDZ is affected only by data movement instructions. RDZ is set if the least significant nibble (four bits) of the result is 0000, regardless of the setting of DCM. If the most significant four bits are not 0000, RDZ is cleared.

Status information is based on the entire single or multi-byte quantity that is processed. The figure below illustrates status on a three-byte quantity.



MULTI-BYTE STATUS

All multi-byte operations except right shift start execution with the least significant byte. All status flags except LSB, RDZ, and DCM are updated after each byte of an operation, and therefore will be correct whenever the memory boundary is reached. The LSB and RDZ flags are set only for the first byte.

For a shift right instruction, where the shift is from the most significant byte to the least significant, the MSB and LDZ flags are set only for the most significant byte; the rest are updated after each byte.

For a complete list of all CPU instructions and their relationships to status indicators, refer to section 4 and appendix C.

NOTES

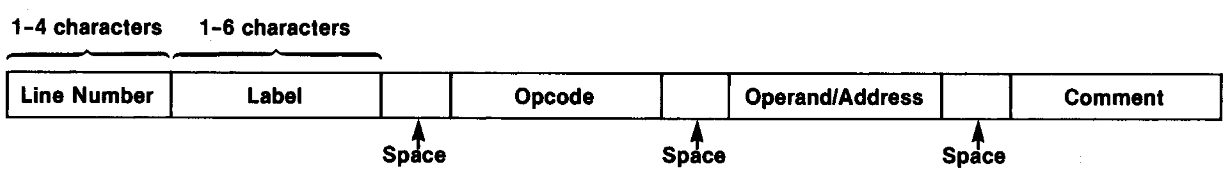
ASSEMBLER INSTRUCTIONS

The HP-83/85 Assembler instructions can manipulate data in the HP-83 or HP-85 central processing unit, and through the CPU, in HP-83/85 RAM as well.

Assembler instructions are of two types: Instructions and pseudo-instructions. Instructions operate directly on the CPU and during assembly are translated directly into machine language object instructions. They are specified by means of opcodes. Pseudo-instructions are entered in the same way as CPU instructions, but they are actually messages to the Assembler ROM. They are specified by means of pseudo-opcodes.

ENTERING INSTRUCTIONS AND PSEUDO-INSTRUCTIONS

Source code is typed into the CRT by entering the line number, followed by a label (if any), followed by the opcode, followed by the address or operand, if required, followed by a comment (if any). When [END LINE] is then pressed, the line is parsed and the elements are assigned to their respective fields on the CRT.



SOURCE CODE INSTRUCTION FORMAT

In assembler mode, the HP-83/85 is sensitive to spacing among the elements of a line of source code. For example:

A statement entered to the CRT as:

```
6Ø LBL LDMD R70,R40
7Ø Label jsb=numval
8Ø  PUBD R52,+R12
9Ø  PUBD 52,+12
1ØØ  CLB R40 !THIS IS A COMMENT
```

After parsing appears as:

```
60 LBL    LDMD R70,R40
70 Label  JSB =NUMVAL
80        PUBD R52,+R12
90        PUBD R52,+R12
100        CLB R40
!THIS IS A COMMENT
   ↑      ↑        ↑
Label   Opcode   Operand or
Field   Field    Address Field
```

LINE NUMBERING

Each line of binary program source code must begin with a line number. These line numbers may be entered individually, or automatic line numbering may be specified with the [AUTO] key.

These line numbers are useful for entering and editing a binary program, but do not correspond to the addresses of the machine language object code that is generated during assembly.

LABELS

No spaces or one space may be typed between the line number and the label field. A label is optional, and may be from one to six characters. A label cannot have a digit as the first character, nor a space as any character; one or more spaces denote the end of the label.

When a label has been entered and parsed, it appears in a label field on the CRT or printer. This field begins in the second character space to the right of the line number.

OPCODES AND PSEUDO-OPCODES

The opcodes and pseudo-opcodes for assembly language instructions may be entered after typing at least two spaces after the line number or at least a single space after a label. Entries in the opcode field are restricted to valid instructions and pseudo-instructions. Blanks are not allowed within the opcode field.

When an opcode or pseudo-opcode has been entered and parsed, it begins in the field nine spaces to the right of the line number.

Opcodes (but not pseudo-opcodes) may be either single-byte (specified by a "B") or multi-byte (specified by an "M").

OPERANDS OR ADDRESSES

Depending upon the format of the instruction, the operand or address field may specify one or more of the following:

--Data Register. A CPU register which may signify single-byte or multi-byte operation.

--Operand. May be a CPU register or a memory location. Depending on the addressing mode, memory can be addressed immediately, indirectly, or by an index.

--Register Pointer. Constant used to load ARP or DRP.

--Label. A label to specify an address or constant.

--Nothing. Some instructions do not require an entry in this field.

An AR or DR in the CPU is specified by an "R" before the register number (e.g., R32), or by an "X" before the register number when indexed addressing is used. The "R" may be omitted when CPU register numbers are typed, since the assembler inserts a missing "R" automatically. The "X" must be typed to indicate register numbers for indexed operations.

COMMENTS

A comment or remark must begin with an exclamation point. A comment must be typed beginning in the first or second space after the line number, or beginning one or more spaces after the other elements of the line of source code.

After being parsed, a comment which has been entered immediately following the other elements of the line begins in column 33; thus, on the HP-83/85 CRT it appears on the following line. A peripheral printer with a column width greater than 32 can permit a comment to appear on the same line as the source code statement.

NUMERIC VALUES

Numeric values can be entered in octal, BCD or decimal notation. A BCD value is entered by immediately following the value with a "C," while a decimal value is followed by a "D;" otherwise the assembler assumes octal values.

Example: LDM R45,=31, 19C, 25D Loads the same bit pattern into registers R45, R46 and R47.

Registers can be specified by octal values only.

SYNTAX AND SYMBOLS USED

The following shows the syntax guidelines once again and also includes a list of the symbols used in the descriptions of assembler instructions.

LDB	Instructions shown in capital letters, but not underlined, must be entered exactly as shown (in either upper-case or lower-case letters).
___	Items shown underlined (e.g., <u>DR</u>) are expressions or names that must be specified in the instruction, statement, or command.
[ ]	Items shown between brackets are optional. (e.g., CMB[D] indicates there is a CMB instruction and also a CMBD instruction available.) If several items are stacked between brackets, any one or none of the items may be specified.
...	Three dots (ellipsis) following a set of brackets indicate that the items between the brackets may be repeated.
←	Is transferred to.
( )	Contents of.
‾‾‾	Complement (e.g., $\overline{x}$ is complement of x). This is one's complement if DCM=0 and nine's complement if DCM=1.

B/M Single-byte or multi-byte instruction.

AR Address register location--location of first byte addressed by ARP. Can be a register (e.g., R32), R* or R#.

DR Data register location--location of first byte addressed by DRP. Can be a register (e.g., R32), R* or R#.

A Address mode for load/store. Can be blank (for immediate), D (for direct), or I (for indirect).

ARP Address Register Pointer. A 6-bit register used to point to one of 64 CPU registers. The byte to which ARP points is often used as the first of two consecutive bytes forming a memory address.

DRP Data Register Pointer. A 6-bit register used to point to one of 64 CPU registers. The location to which DRP points is often used as the destination for data loaded into the CPU.

R(x) CPU register addressed by (x).

M(x) Memory location addressed by (x). (x) must be a 16-bit address.

PC Program Counter. CPU registers R4 and R5. Used to address the instruction being executed.

SP Subroutine Stack Pointer. CPU registers 6 and 7. Used to point to the next available location on the subroutine return address stack.

EA Effective Address. The location from which data is read for load-type instructions or the location where data is placed for store-type instructions.

ADR Address. The two-byte quantity directly following an instruction that uses the literal direct, literal indirect, index direct or index indirect addressing mode. This quantity is always an address.

The following pages show the HP-83/85 Assembler ROM instructions that are used to manipulate the CPU and external memory. These instructions are illustrated in an abbreviated form in this section; for a complete list of all forms of each instruction, refer to appendix C.

Also contained in this section are the Assembler ROM pseudo-instructions.

LOAD/STORE INSTRUCTIONS

The instructions for loading and storing data have access to all eight addressing modes, and they can be single-byte or multi-byte.

LD

CPU Instruction

Load

Format: LDBA DR, operand — Single byte
LDMA DR, operand — Multi-byte

Operation: DR←(EA)

Description: Data register is loaded with the contents of the effective address determined by the operand and the addressing mode.

ST

CPU Instruction

Store

Format: STBA DR, operand — Single byte
STMA DR, operand — Multi-byte

Operation: (DR)→EA

Description: Contents of data register are stored in effective address determined by the operand and the addressing mode.

ADDRESSING MODES

The HP-83/85 CPU allows for several addressing modes. These include literal, register, indexed and stack modes of memory access.

Not all addressing modes are available to all instructions. The load (LD) and store (ST) instructions have access to all addressing modes except stack addressing, and they are used here for illustration. For a list of the addressing modes available to any particular instruction, consult the description of that instruction in this section or in appendix C.

In addressing, all addresses are referred to as two-byte quantities. Because all addresses are two consecutive bytes, only the first byte of the sequence is referenced. For instance, the AR is actually a single byte within the CPU register bank that is pointed to by the ARP. When the AR is described as being an address, remember that R (ARP) contains the low byte of the address and R (ARP + 1) contains the upper byte of the address.

The multi-byte feature of the CPU allows data to be manipulated in quantities of from one to eight bytes. Therefore, in the following descriptions, only the address of the first byte of data is specified. As explained earlier, the number of bytes is determined by the distance of the DR from the next consecutive boundary.

In the following descriptions, the effective address (EA) points to the first byte of data to be loaded for load instructions.

For store instructions, EA points to the location where the first byte of data is stored.

REGISTER MODE

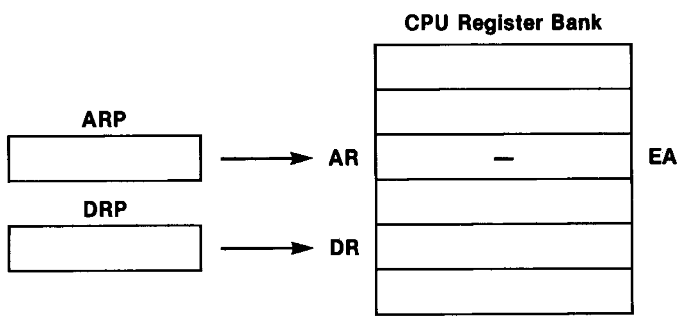
The first category of addressing is the <u>register</u> addressing mode. This mode allows the CPU registers (64_{10} bytes) to be used as addresses as well as for data. There are three levels of register addressing modes.

REGISTER IMMEDIATE

Format: Opcode B/M DR, AR

Effective Address: AR

Description: The operand is another CPU register (single or multi-byte) beginning at AR. Thus, the AR is the source for load instructions or the destination for store instructions.



REGISTER IMMEDIATE ADDRESSING

Examples: LDB R36, R32 Loads contents of R32 into CPU register R36.

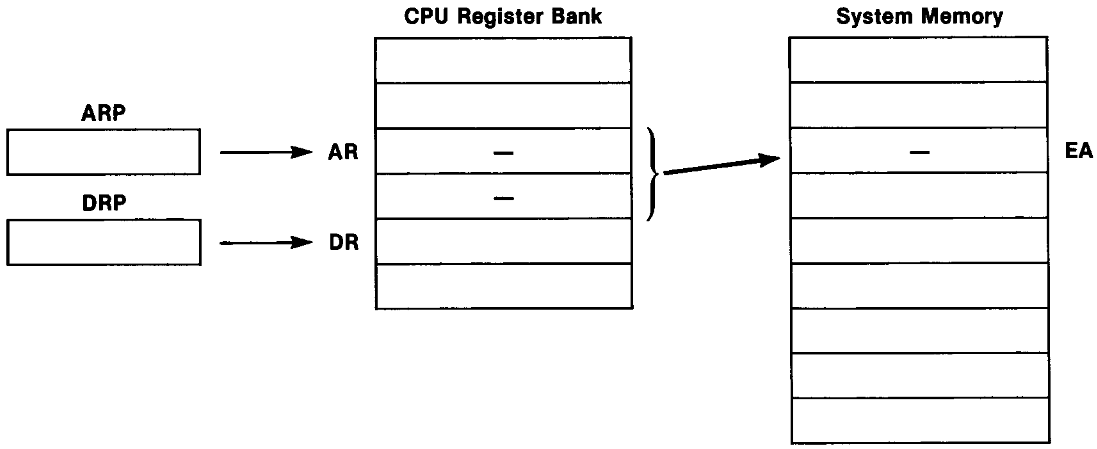
STM R40, R50 Stores contents of registers R40 through R47 into registers R50 through R57.

REGISTER DIRECT

Format: Opcode B/M D DR, AR

Effective Address: M(AR)

Description: The effective address is a location in system memory that is addressed by the AR. This mode is useful when using a CPU register as a pointer to system memory.



REGISTER DIRECT ADDRESSING

Examples: LDBD R36, R32 Loads CPU register R36 with the contents of the system memory location addressed by R32-R33.

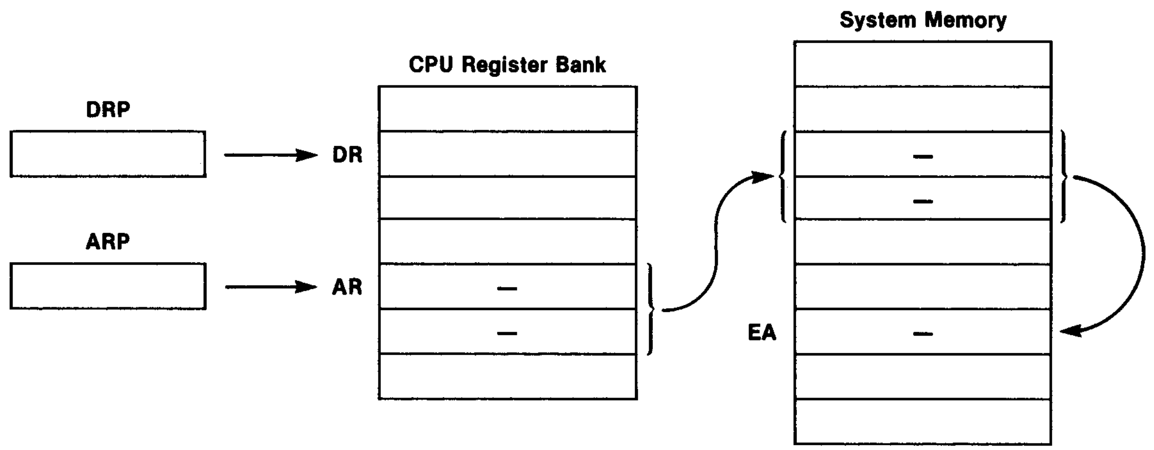
STMD R40, R50 Stores contents of R40-R47 into system memory beginning with location addressed by R50-R51.

REGISTER INDIRECT

Format: Opcode B/M I DR, AR

Effective Address: M(M(AR))

Description: The address register points to a system memory location, which in turn points to another memory location that is the effective address.



REGISTER INDIRECT ADDRESSING

Example: LDBI R36, R32 If R32 and R33 contain the address 105371, loads CPU register R36 with the contents of the memory location that is addressed by the contents of system memory locations 105371 and 105372.

LITERAL MODE

The second of the categories of address modes is the literal mode. In literal mode, the operand is a literal quantity stored in memory immediately following the opcode. A literal string can be:

--BCD constant, e.g., 99C, ..., 79C ($\leq 10_8$ bytes)

--Octal constant, e.g., 12, ..., 277 ($\leq 10_8$ bytes)

--Decimal constant, e.g., 201D, ..., 9D ($\leq 10_8$ bytes)

--Label (The literal quantity is a one- or two-byte value or address assigned to the label.)

The programmer is responsible for ensuring that the number of bytes of the literal string matches the DRP setting. The assembler does not check for mismatch.

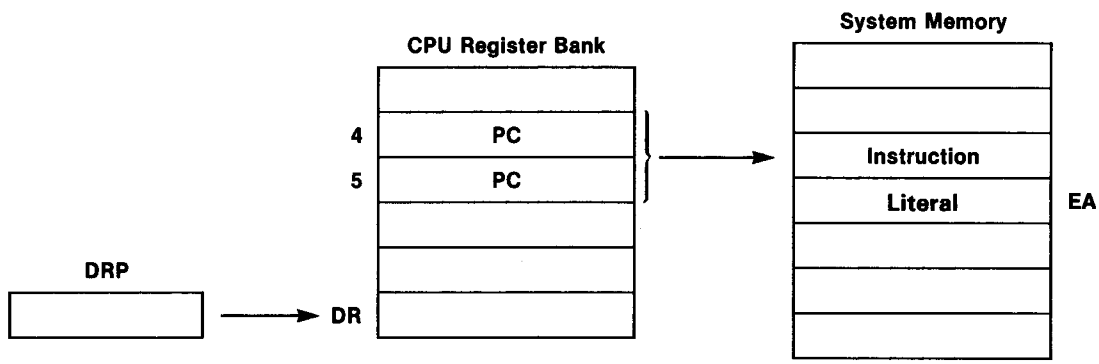
There are three types of literal addressing modes.

LITERAL IMMEDIATE

Format: Opcode B/M DR, = literal

Effective Address: (PC+1)

Description: The operand is a literal string that, during assembly, is stored in memory immediately after the instruction opcode. This mode is useful for loading constants into the CPU register bank.



LITERAL IMMEDIATE ADDRESSING

Examples: LDB R36, = 3D Loads 3_{10} into CPU register R36.

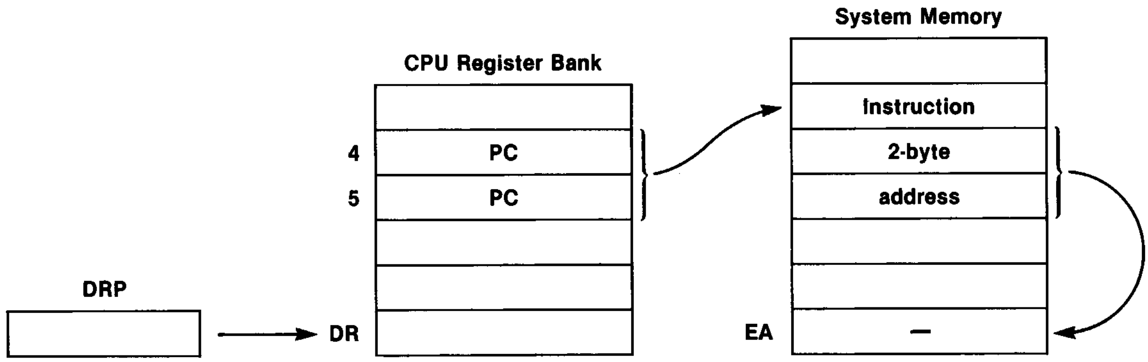
LDM R40, = 0,0,0,0,0,0,0,120 Loads 120_8 (i.e., a floating-point 5) into registers R40-R47.

LITERAL DIRECT

Format: Opcode B/M D DR, = label

Effective Address: M(PC+1)

Description: The operand is a memory location that, after assembly, is addressed by a two-byte literal quantity stored immediately after the instruction opcode. The label defines the two-byte literal quantity to be used by the Assembler ROM.



LITERAL DIRECT ADDRESSING

Examples: LDBD R34, = ROMFL Loads the contents of the memory location addressed by the label ROMFL into CPU register R34.

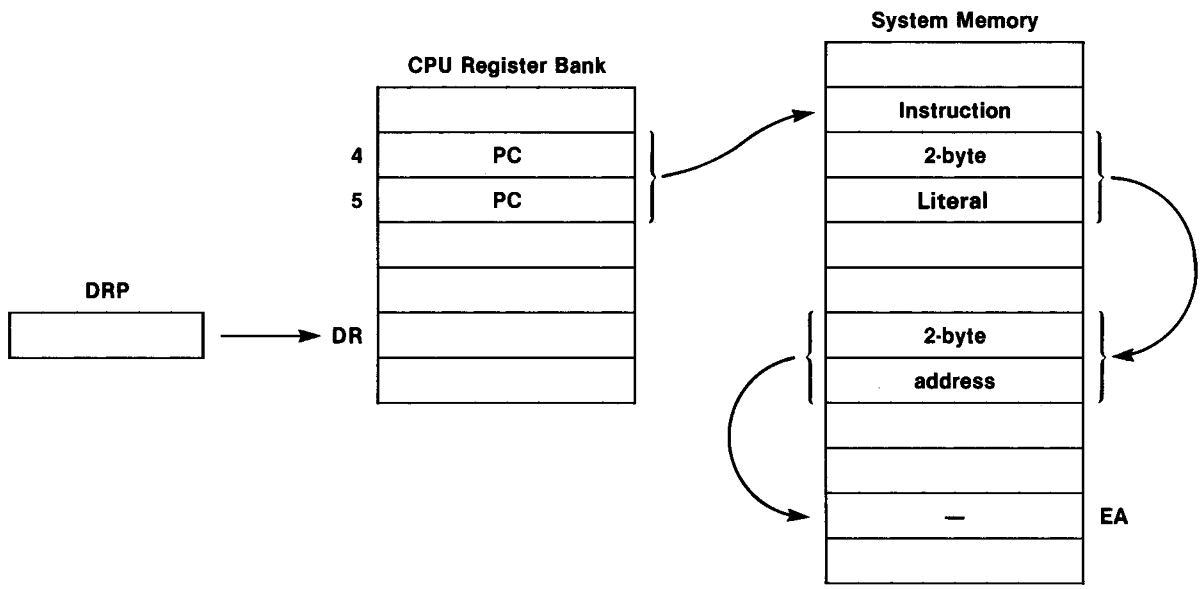
STMD R74, = CHIDLE Stores contents of CPU registers R74 through R77 into four memory locations beginning with the location addressed by the label CHIDLE.

LITERAL INDIRECT

Format: Opcode B/M I DR, = label

Effective Address: M(M(PC+1))

Description: The operand is a memory location that, after assembly, is addressed by a two-byte memory location that itself is addressed by a two-byte literal quantity stored immediately after the instruction opcode. The label defines the two-byte literal quantity used by the Assembler ROM.



LITERAL INDIRECT ADDRESSING

Example: STBI R30, = ADDR Stores the contents of CPU register R30 into the memory location addressed by another memory location which is itself addressed by the two-byte literal quantity specified by the label ADDR.

INDEX MODE

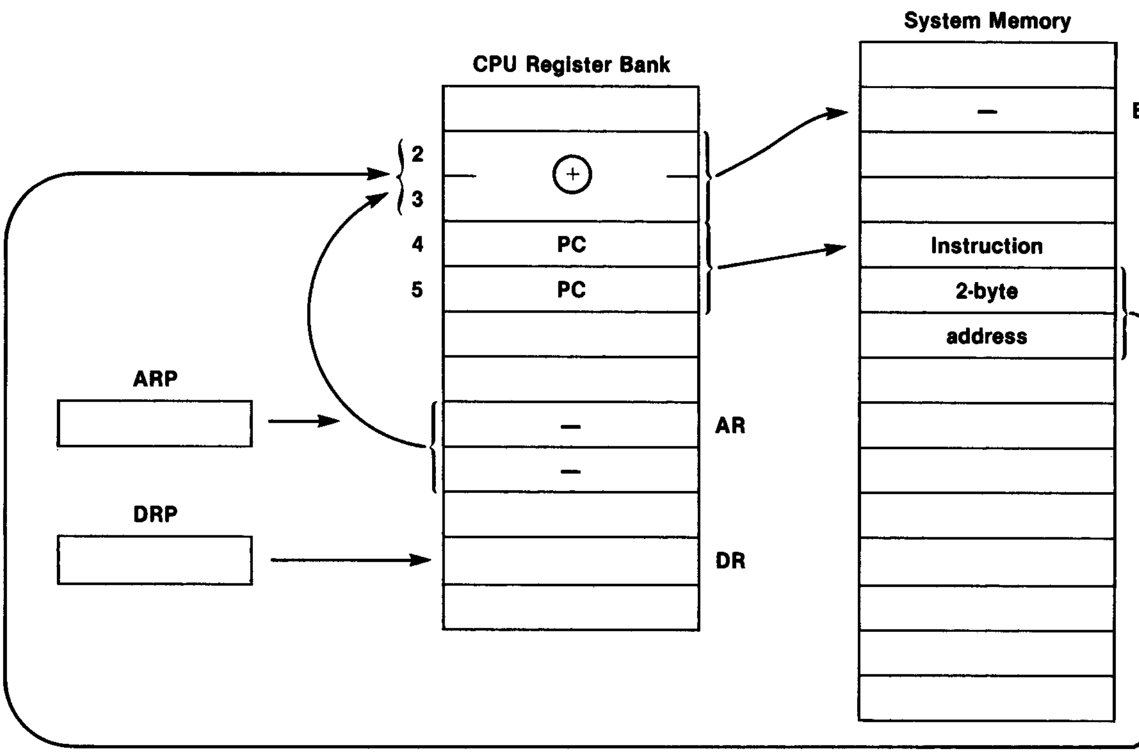
The index mode is the third addressing category. Indexing is useful for accessing data when the data is stored in a table. In indexed addressing, a fixed base address is added to an offset to create the desired address. The CPU performs this addition using CPU registers 2 and 3. After an index instruction, registers 2 and 3 contain the effective address (i.e., the sum of the base and the offset). Neither the original base nor the offset is altered in memory. There are two modes for indexed addressing.

INDEX DIRECT

Format: Opcode B/M D DR, XAR, label

Effective Address: M(AR+(PC+1))

Description: The effective address is found by adding (in binary) the two-byte contents of the AR to the two-byte address that immediately follows the instruction opcode in memory.



INDEXED DIRECT ADDRESSING

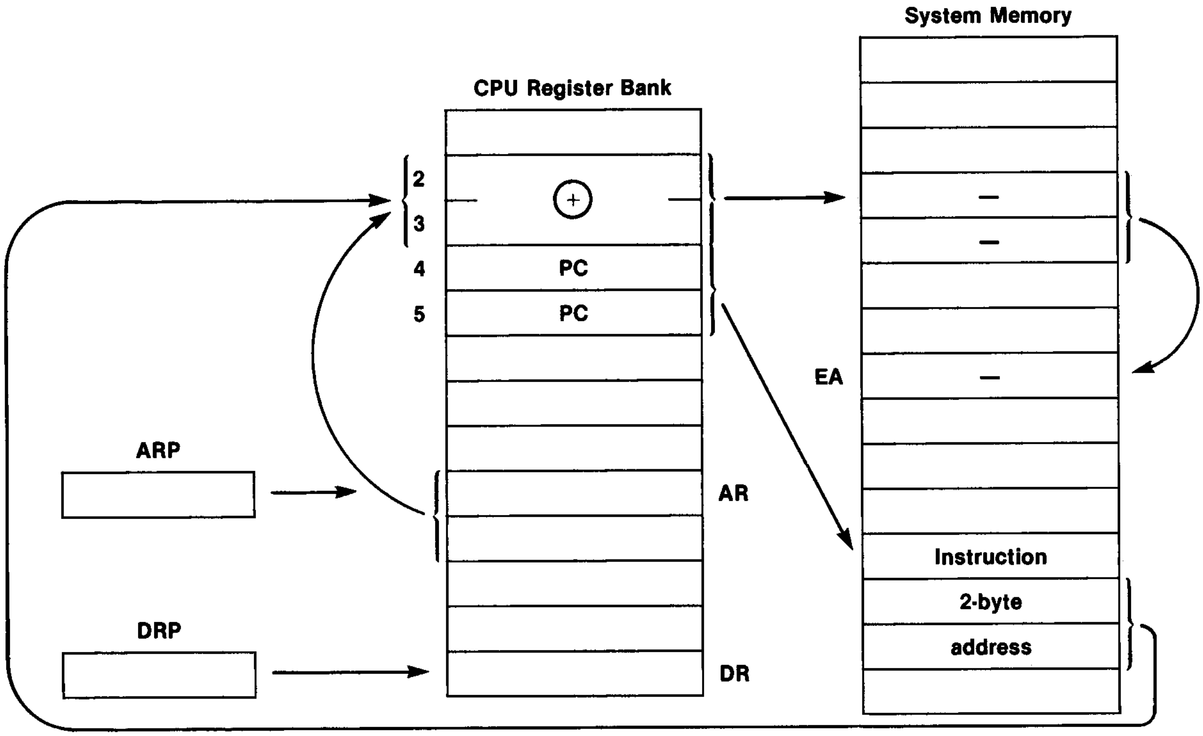
Example: LDBD R36, X30, TABLE Loads into CPU register R36 the contents of the memory location addressed by registers R2 and R3. R2 and R3 contain the sum of the contents of R30 and the contents of the address TABLE.

INDEX INDIRECT

Format: <u>Opcode</u> <u>B/M</u> I <u>DR</u>, <u>XAR</u>, <u>label</u>

Effective Address: M(M(AR+(PC+1)))

Description: The effective address is found in a memory location. This memory location is found by adding (in binary) the two-byte contents of the AR to the two-byte address that immediately follows the instruction opcode in memory. This mode is useful when addresses are stored in table form.



INDEXED INDIRECT ADDRESSING

Example: STMI R36, X30, OFFST Stores the contents of CPU register R36 and R37 in memory, beginning with the location addressed by another memory location which is itself addressed by CPU registers 2 and 3. Registers 2 and 3 contain th sum of the address in R30 plus the offset specified by the label OFFST.

STACK INSTRUCTIONS

There is a large set of instructions that are available to push data onto and pop data from stacks in the main memory of the HP-83/85. These stacks can be addressed by the instructions using direct or indirect addressing.

PU
Push

CPU Instruction

Format: PUB D/I DR +/- AR Push single byte
PUM D/I DR +/- AR Push multi-byte

Description: Pushes single byte or multi-byte onto stack. D/I indicates direct or indirect addressing. +/- indicates stack pointer is incremented (increasing stack) or decremented (decreasing stack) in memory.

Examples:
```
PUBD R32, +R12
PUBI R32, -R46
```

PO
Pop

CPU Instruction

Format: POBD/I DR +/- AR Pop single byte
POMD/I DR +/- AR Pop multi-byte

Description: Pops single byte or multi-byte off stack. D/I indicates direct or indirect addressing. +/- indicates stack pointer is incremented (increasing stack) or decremented (decreasing stack) in memory.

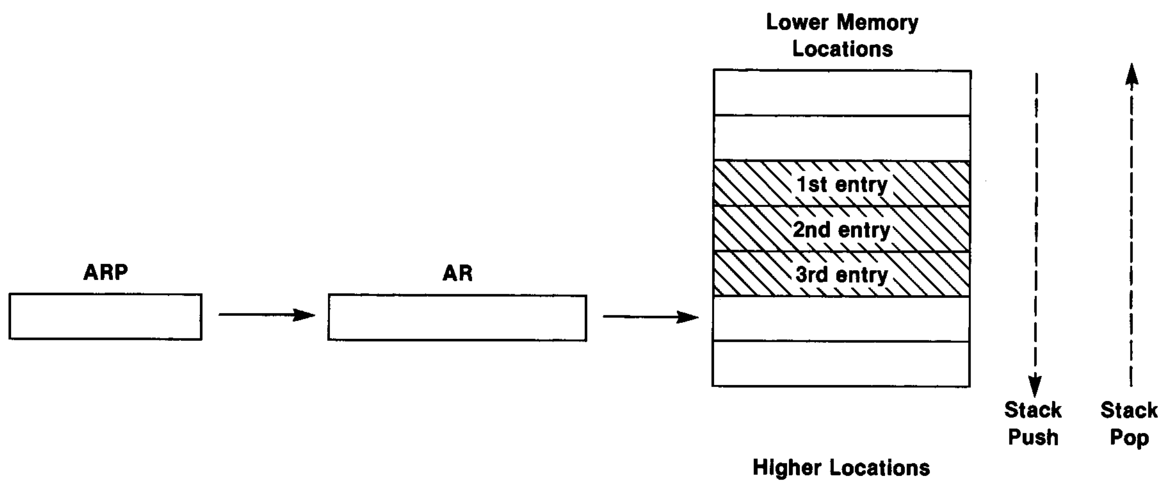
STACK ADDRESSING

CPU registers R6 and R7 are permanently dedicated, and always contain the address of the subroutine return stack. CPU registers R12 and R13 contain, by convention, the address of the operational stack used during runtime by many of the internal HP-85 routines. The user can, of course, address a stack from nearly any CPU register pair.

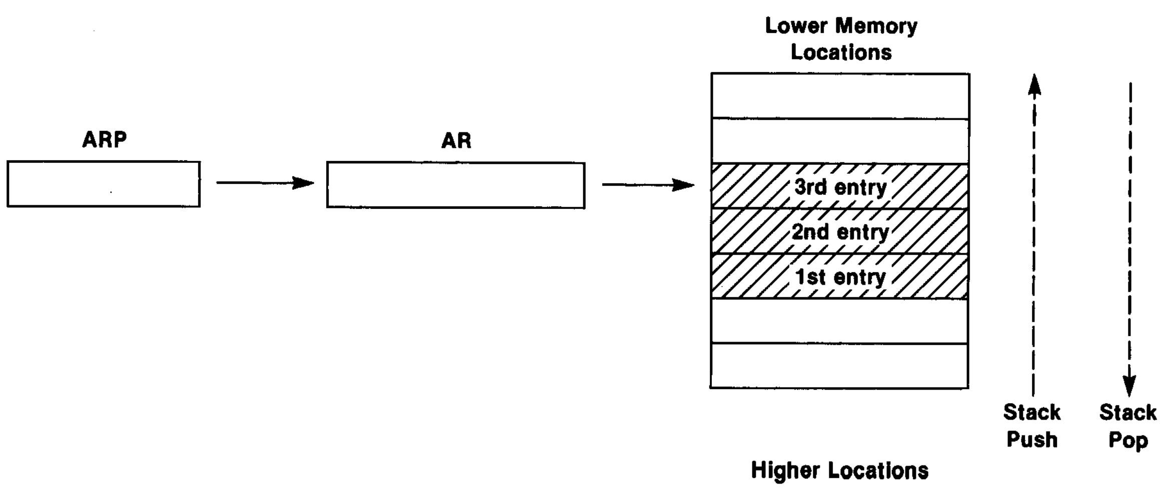
Stacks may be increasing or decreasing. An increasing stack is one which is filled in the direction of higher memory locations and from which data is removed in the direction of lower memory locations. In a decreasing stack, data is pushed in the direction of lower memory locations, and taken off in the direction of higher memory locations. To avoid confusion, it is best to address a particular stack using only instructions for an increasing stack or only instructions for a decreasing stack, but not both.

For stack addressing, the stack pointer is contained in the AR. Multiple stacks are handled by having multiple stack pointers within the CPU register space. A stack is activated by setting ARP equal to the location of that stack's pointer.

For an increasing stack, the AR must point to the next available location on the stack. For a decreasing stack, the AR points to the occupied location on top of that stack.



INCREASING STACK



DECREASING STACK

STACK DIRECT

In this addressing mode, the stack is presumed to contain data. Stores to the stack (pushes) fill the stack. Loads from the stack (pops) empty the stack.

For a push onto an increasing stack, the AR points to the location where data is to be stored. Following the store, the AR is incremented by the number of bytes stored. For a pop operation from an increasing stack, the AR is first decremented by the number of bytes to be popped off. The AR then points to the location of the data to be removed from the stack.

For a pop from a decreasing stack, the AR points to the location of the data to be removed. Following the removal, the AR is incremented by the number of bytes moved. For a push operation onto a decreasing stack, the AR is first decremented by the number of bytes to be stored on the stack. Then the data is pushed onto the stack.

STACK INDIRECT

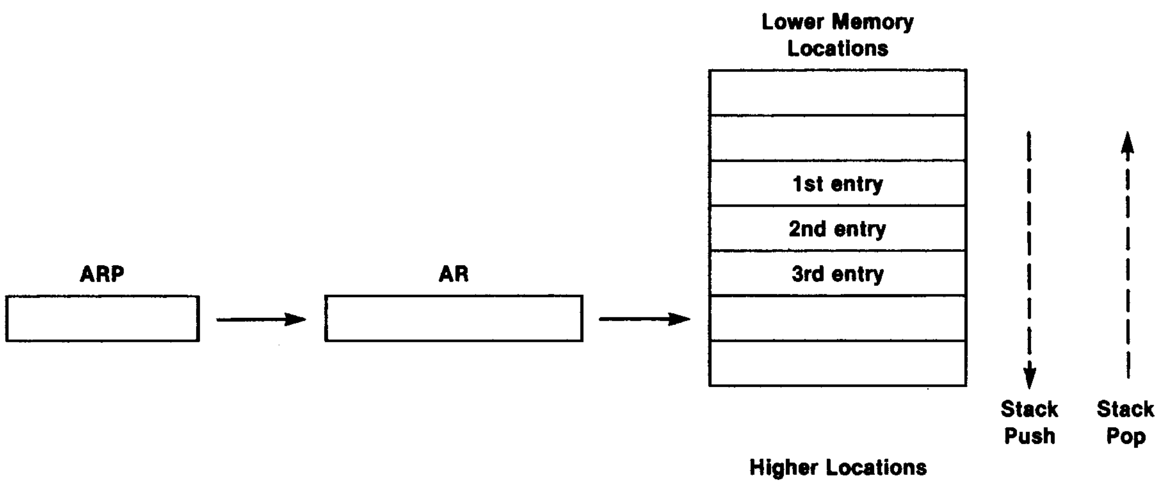
In this addressing mode, the stack is presumed to contain an ordered list of addresses. These addresses point to the location from which data is read by pops or to the location into which data is stored by pushes.

For a push onto an increasing stack, the AR points to the effective address. After storing data in M(EA), the AR is incremented by two. For a pop instruction from an increasing stack, the AR is first decremented by two in order to point to the effective address. M(EA) is then loaded into the CPU register designated by the DRP.

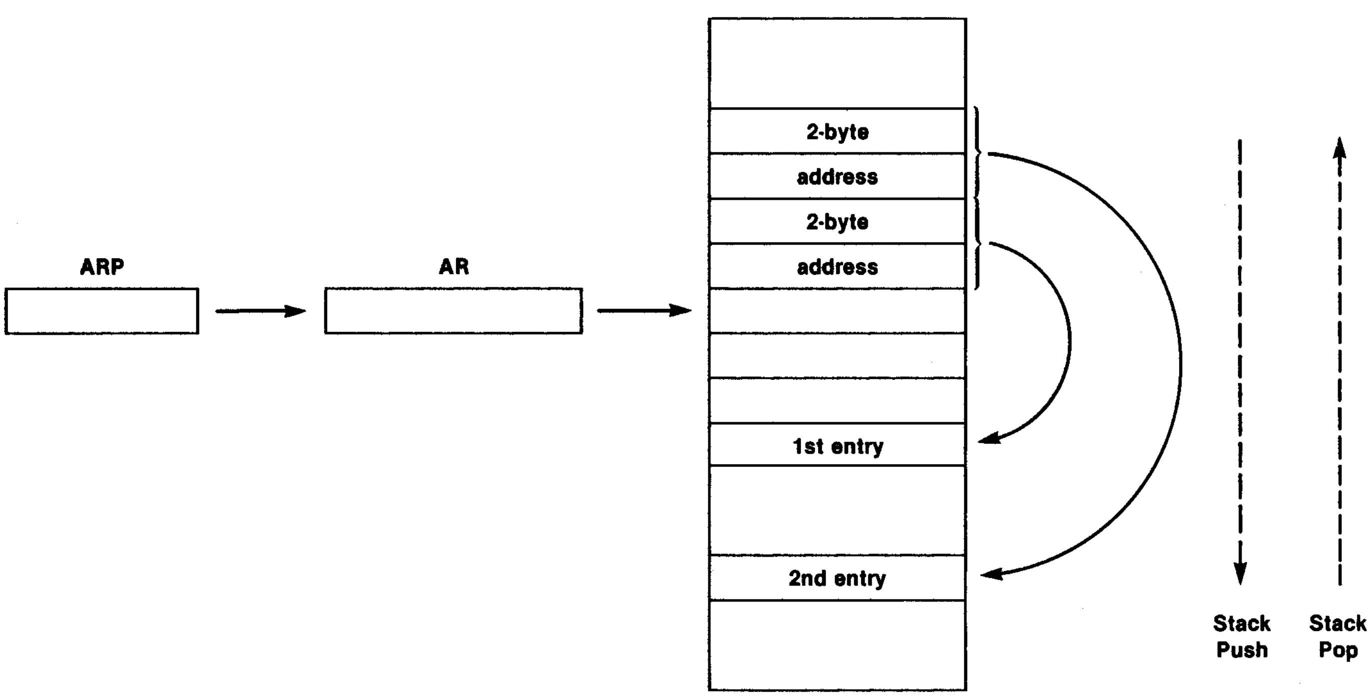
INSTRUCTIONS FOR AN INCREASING STACK

An increasing stack is one which is pushed in the direction of higher addresses (+) and popped in the direction of lower addresses (-).

D (Direct Mode)



I (Indirect Mode)



Each entry can be one or more bytes

INCREASING STACK

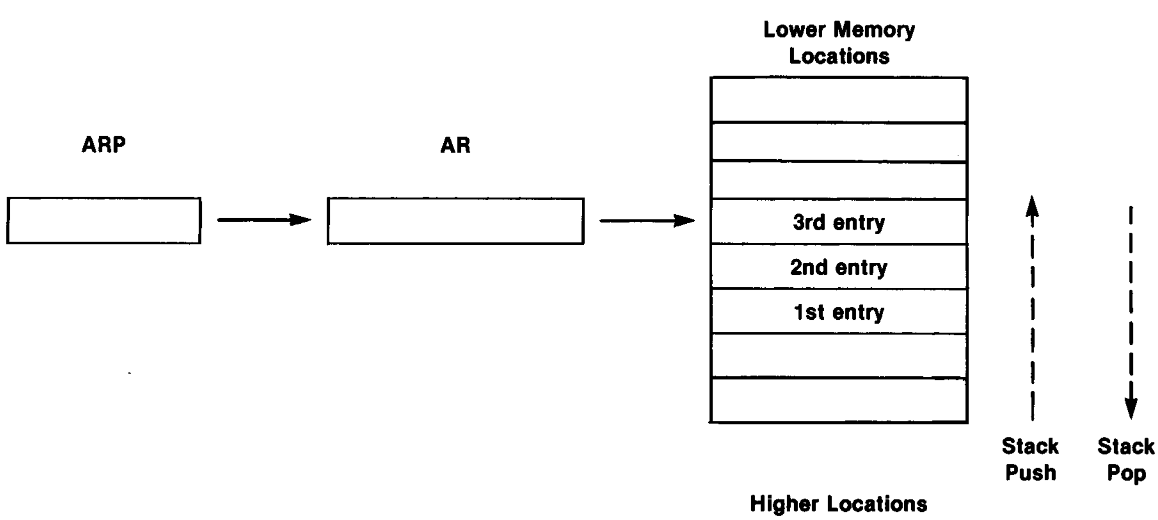
The instructions available for use with an increasing stack are:

PUBD	DR, +AR	Push byte direct with increment
PUMD	DR, +AR	Push multi-byte direct with increment
PUBI	DR, +AR	Push byte indirect with increment
PUMI	DR, +AR	Push multi-byte indirect with increment
POBD	DR, -AR	Pop byte direct with decrement
POMD	DR, -AR	Pop multi-byte direct with decrement
POBI	DR, -AR	Pop byte indirect with decrement
POMI	DR, -AR	Pop multi-byte indirect with decrement

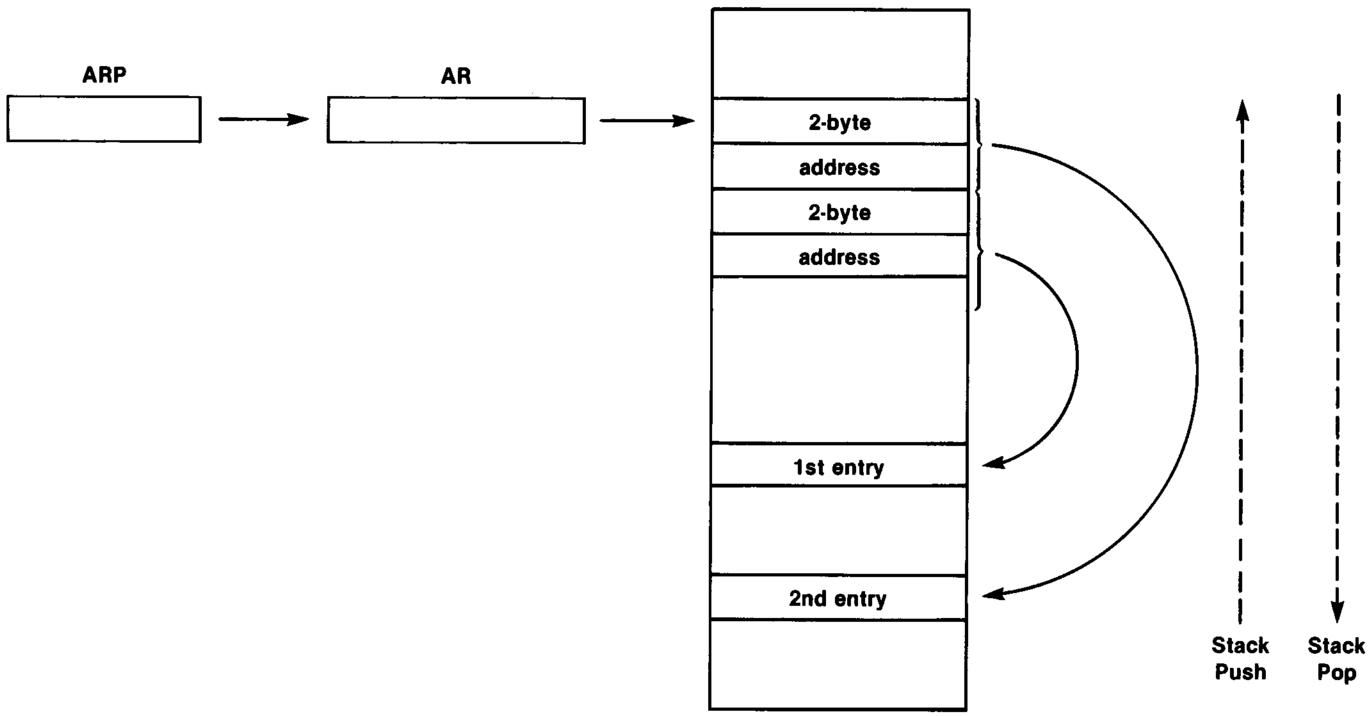
INSTRUCTIONS FOR A DECREASING STACK

A decreasing stack is one which is pushed in the direction of lower addresses (-) and popped in the direction of higher addresses (+).

D (Direct Mode)



I (Indirect Mode)



Each entry can be one or more bytes

DECREASING STACK

The instructions available for use with a decreasing stack are:

PUBD DR, -AR	Push byte direct with decrement
PUMD DR, -AR	Push multi-byte direct with decrement
PUBI DR, -AR	Push byte indirect with decrement
PUMI DR, -AR	Push multi-byte indirect with decrement
POBD DR, +AR	Pop byte direct with increment
POMD DR, +AR	Pop multi-byte direct with increment
POBI DR, +AR	Pop byte indirect with increment
POMI DR, +AR	Pop multi-byte indirect with increment

ARITHMETIC AND LOGICAL INSTRUCTIONS

The arithmetic and logical instructions consist of add, subtract, compare, logical AND and logical OR instructions.

AD

CPU Instruction

Add

Format: ADB [D] DR, operand — Add byte
ADM [D] DR, operand — Add multi-byte

Operation: DR ← DR + operand

Description: Add single or multi-byte. The contents of the effective address determined by the addressing mode are added to the DR. If DCM=1, BCD addition is performed; otherwise, binary addition is performed. The result is stored in the data register.

Examples:

```
ADB R40, R50
ADMD R30,=LABEL
```

ANM

CPU Instruction

Logical AND

Format: ANM [D] DR, operand

Operation: DR ← DR · operand

Description: The DR is loaded with the logical AND of itself and the contents of the effective address determined by the addressing mode used. This instruction is multi-byte only.

Examples:

```
ANM R40, R50
ANMD R32,=LABEL
```

CM

CPU Instruction

Compare

Format: CMB [D] DR, operand — Compare byte
CMM [D] DR, operand — Compare multi-byte

Operation: DR + ten's complement of operand if BCD mode set
DR + two's complement of operand if binary mode set

Description: Compares operand with data register(s). The contents of the effective address determined by the operand and the addressing mode are subtracted from DR. BCD subtraction is performed if DCM=1; otherwise a binary subtraction is performed. The result is used to affect CPU status indicators and is not stored; DR is not affected.

Examples:
```
CMB R24,=377
CMM R22, R32
```

OR

CPU Instruction

Logical OR (Inclusive)

Format: ORB DR, AR — Inclusive OR (single byte)
ORM DR, AR — Inclusive OR (multi-byte)

Operation: DR ← DR v AR

Description: Contents of DR are replaced with inclusive OR of DR and AR. CY and OVF are cleared.

Examples:
```
ORB R21, R41
ORM R40, R70
```

SB

CPU Instruction

Subtract

Format: SBB [D] DR, operand — Subtract byte
SBM [D] DR, operand — Subtract multi-byte

Operation: DR ← DR + ten's complement of operand if BCD mode
DR ← DR + two's complement of operand if binary mode

Description: The contents of the effective address determined by the addressing mode and the operand are subtracted from the contents of the DR. BCD subtraction is performed if DCM=1; otherwise binary subtraction is performed. The result is stored in DR. CY is set if the result is positive, cleared if the result is negative.

Example: SBM R26,=177, 0

XR

CPU Instruction

Logical OR (Exclusive)

Format: XRB DR, AR — Exclusive OR (single byte)
XRM DR, AR — Exclusive OR (multi-byte)

Operation: DR ← DR ⊕ AR

Description: Contents of DR are replaced with the exclusive OR of DR and AR. CY and OVF are cleared.

Example: XRM R40, R50

SHIFT INSTRUCTIONS

All shift instructions can be BCD or binary. The shift instructions consist of logical left, logical right, extended left and extended right instructions; all are available in single byte or multi-byte modes.

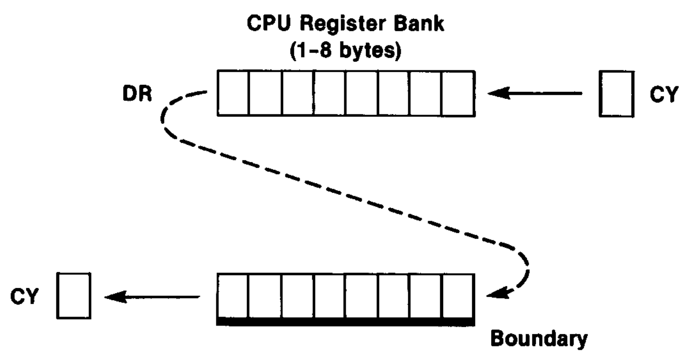
EL

CPU Instruction

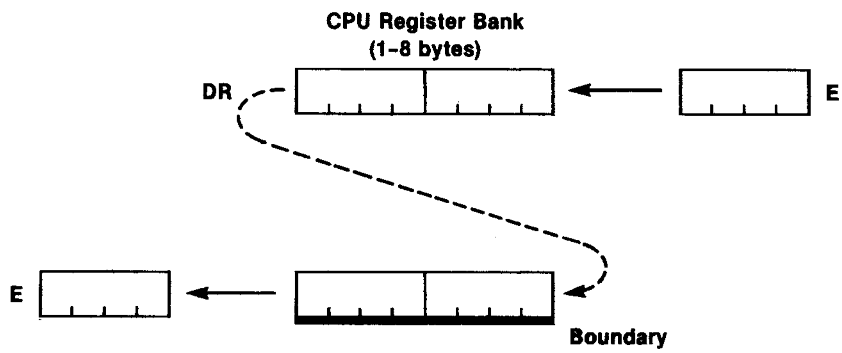
Extended Left Shift

Format:	ELB DR	Extended left shift byte
	ELM DR	Extended left shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted left one bit position. Carry flag CY is loaded from MSB. LSB is loaded from CY. OVF is set if the shift causes a sign change.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted left one digit position (i.e., four bits) through the E register. CY is cleared.



ER

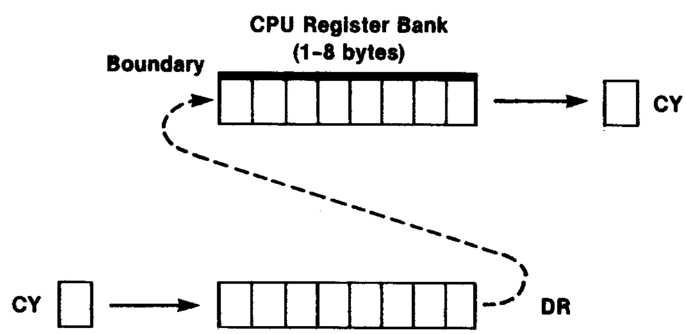
CPU Instruction

Extended Right Shift

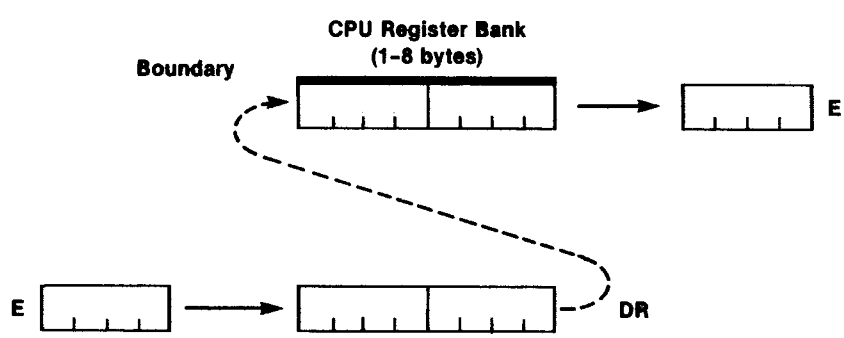
Format: ERB DR — Extended right shift byte

ERM DR — Extended right shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted right one bit position. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. Carry flag CY is loaded from LSB. MSB is loaded from CY.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted right one digit position (i.e., four bits) through the four-bit E register. CY is cleared.



Notice that a multi-byte right shift instruction, unlike other multi-byte instructions, proceeds from the DR to the preceding (i.e., lower-numbered) boundary.

Example: ERM R47 Shifts all eight bytes of R40 - R47 right.

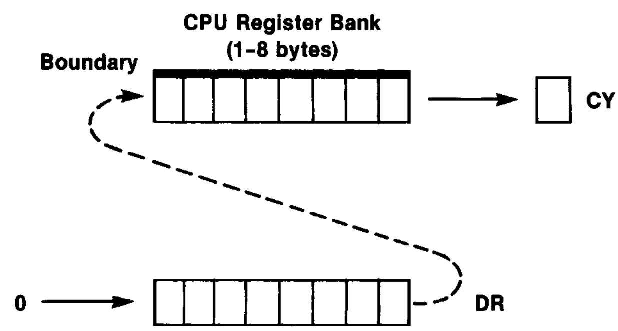
LR

CPU Instruction

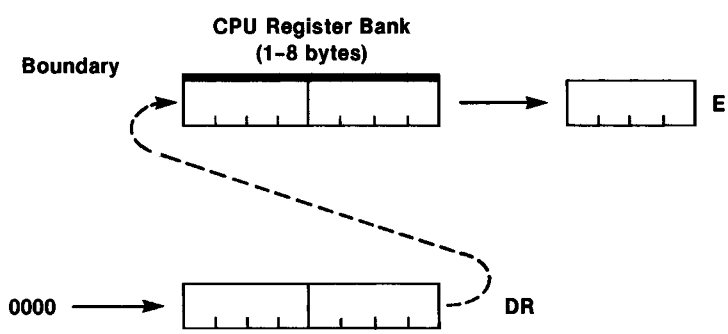
Logical Right Shift

Format: LRB DR Logical right shift byte
LRM DR Logical right shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR (one to eight bytes) are shifted right one bit position, and the MSB is cleared. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. Carry flag CY is loaded from LSB.



BCD Mode. In BCD mode, the contents of DR (one to eight bytes) are shifted right one digit position (i.e., four bits), and the most significant digit is cleared. For multi-byte shifts, the shift proceeds from DR to the next lower boundary. The least significant digit is shifted into the four-bit E register.



Notice that a multi-byte right shift instruction, unlike other multi-byte instructions, proceeds from the DR to the preceding (i.e., lower-numbered) boundary.

Example: LRM R54 Shifts contents of R54, R53, R52, R51, and R50 right.

LL

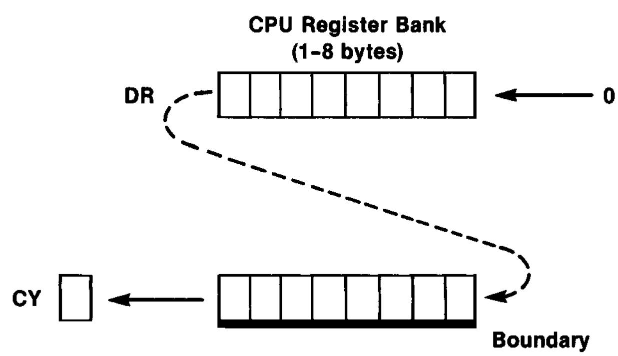
CPU Instruction

Logical Left Shift

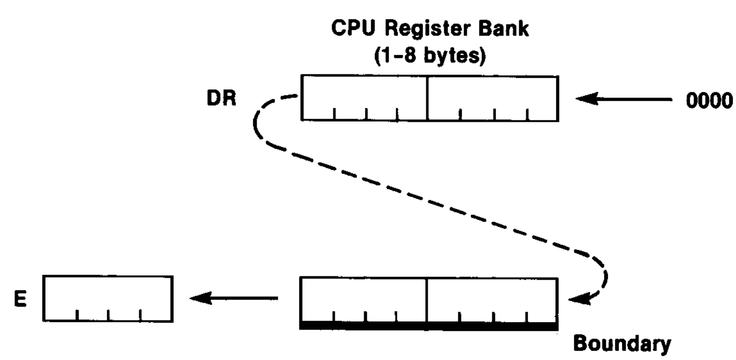
Format:

LLB DR	Logical left shift byte
LLM DR	Logical left shift multi-byte

Description: Binary Mode. In binary mode, the contents of DR are shifted left one bit position, and the LSB is cleared. The bit shifted out of MSB is saved in CY. OVF is set if the shift causes a sign change.



BCD Mode. In BCD mode, the contents of DR are shifted left one digit position (i.e., four bits), and the least significant digit is cleared. The digit shifted out of the most significant digit position is saved in the E register. CY is cleared.



Example: LLM R45 Shifts contents of R45, R46, and R47 left one bit position through CY (in binary mode) or left one digit position through E (in BCD mode).

REGISTER INCREMENT AND DECREMENT INSTRUCTIONS

The increment and decrement instructions for the CPU registers can be BCD or binary.

DC

CPU Instruction

Decrement

Format: DCB DR Decrement byte

DCM DR Decrement multi-byte

Operation: DR ← DR + two's complement of 1 (binary mode)

DR ← DR + ten's complement of 1 (BCD mode)

Description: Binary Mode. In binary mode, DR is decremented by 1 (binary). OVF is set if this operation causes a sign to change to a positive value. CY is set by decrementing a non-zero number.

BCD Mode. In BCD mode, DR is decremented by 1 (decimal). OVF is cleared. CY is set by decrementing a non-zero number.

Example: DCB R12

IC

CPU Instruction

Increment

Format: ICB DR — Increment byte
ICM DR — Increment multi-byte

Operation: DR ← DR + 1

Description: Binary Mode. In binary mode, DR is incremented in binary by 1. OVF is set if this operation causes a sign change to a negative value.

BCD Mode. In BCD mode, DR is incremented in decimal by 1. OVF is cleared.

Example: ICM R40

COMPLEMENT INSTRUCTIONS

The complement instructions can be BCD or binary.

NC

CPU Instruction

Nine's (Or One's) Complement

Format: NCB DR — Nine's (or one's) complement byte
NCM DR — Nine's (or one's) complement multi-byte

Operation: DR ← $\overline{DR}$

Description: Binary Mode. In binary mode, the one's complement of the contents of DR replace the contents of DR. CY and OVF are cleared.

BCD Mode. In BCD mode, the nine's complement of the contents of DR replace the contents of DR. CY and OVF are cleared.

Example: NCB R30

TC

CPU Instruction

Ten's (Or Two's) Complement

Format: TCB DR — Ten's (or two's) complement byte

TCM DR — Ten's (or two's) complement multi-byte

Operation: $DR \leftarrow \overline{DR} + 1$

Description: Binary Mode. In binary mode, the two's complement of the contents of DR replaces the contents of DR. CY is set if the contents of DR were zero. OVF is set if contents of DR were 100...000.

BCD Mode. In BCD mode, the contents of DR are replaced with their ten's complement. CY is set if the contents of DR were zero. OVF is cleared.

Example: TCM R50

TEST INSTRUCTION

The test instruction can check the status of single-byte or multi-byte CPU registers.

TS

CPU Instruction

Test

Format: TSB DR Test byte
TSM DR Test multi-byte

Description: The contents of DR are tested and condition flags are set accordingly. CY and OVF are cleared.

Example: TSM R36

REGISTER CLEAR INSTRUCTION

The clear instruction permits the clearing of any byte or of any multi-byte portion of the CPU register bank.

CL

CPU Instruction

Clear

Format: CLB DR Clear byte
CLM DR Clear multi-byte

Operation: DR ← 0

Description: DR is cleared. CY and OVF are cleared.

Example: CLB R47

SUBROUTINE JUMP INSTRUCTION

The subroutine jump instruction is available in the literal direct or the indexed addressing mode.

JSB

CPU Instruction

Jump to Subroutine

Format: JSB = label — Jump subroutine literal direct
JSB XR, label — Jump subroutine indexed

Operation: Literal Direct. M(SP) ← PC+3, SP ← SP+2, PC ← M(PC+1)
Indexed. M(SP) ← PC+3, SP ← SP+2, PC ← AR + M(PC+1)

Description: The PC is saved in the memory location addressed by the R6 stack pointer. Program control is then transferred to the location defined by the label. In indexed addressing, control is transferred to the location defined by the two-byte contents of the address register plus the label.

After a subroutine jump, the next RTN instruction executed causes a return to the instruction after the JSB.

Examples:

```
JSB = LOC1
JSB X32, LOC2
```

Note: Since an indexed subroutine jump (i.e., JSB XR, label) can cause a jump to an unlabeled destination, the programmer must ensure that the ARP and DRP are set to ensure proper operation at the destination. See Handling of ARP and DRP During Assembly later in this section.

CONDITIONAL JUMP INSTRUCTION

The conditional jump instruction can alter execution based on 16 different conditions in the CPU.

J

CPU Instruction

Conditional Jump

Format:

JMP label	Unconditional jump	
JNO label	Jump on no overflow	
JOD label	Jump on odd	
JEV label	Jump on even	
JPS label	Jump on positive	Takes overflow into consideration. (Exclusive OR of MSB and OVF.)
JNG label	Jump on negative	
JZR label	Jump on zero	
JNZ label	Jump on non-zero	
JEZ label	Jump on E zero	
JEN label	Jump on E non-zero	
JCY label	Jump on carry	
JNC label	Jump on no carry	
JLZ label	Jump on left digit zero	
JLN label	Jump on left digit non-zero	
JRZ label	Jump on right digit zero	
JRN label	Jump on right digit non-zero	

Description: This group of instructions gives the capability of branching as a function of status conditions previously generated. The branching capability uses relative addressing. If the status condition interrogated is found to be true, then the relative branch to the address of the label will be taken. Otherwise, the next instructions after the jump will be executed.

Each jump instruction is assembled into two bytes: An opcode, and an offset in two's complement notation.

A jump can cover 400_8 destinations from 200_8 before the next instruction to 177_8 after the next instruction. The address to which the jump is made is the sum of the address of the jump instruction plus the offset plus two.

Example: JMP INITAL When assembled, this instruction would appear as shown below.

200			
.			
.			
.			
375	n	---	Offset = -3
376	n + 1	JMP	Offset = -2
377	n + 2	Offset	Offset = -1 (Current byte)
0	n + 3	---	Offset = 0 (Next byte)
1	n + 4	---	Offset = +1
2	n + 5	---	Offset = +2
.			
.			
.			
177			

ARP AND DRP LOAD INSTRUCTIONS

Two instructions are available for loading the address register pointer or the data register pointer. These instructions are not normally needed because the assembler automatically generates necessary ARPs and DRPs where required.

ARP

CPU Instruction

Load ARP

Format: ARP AR

Operation: ARP

Description: Sets address register pointer to point to address register.

Example: ARP R25 Sets ARP to point to R25.

DRP

CPU Instruction

Load DRP

Format: DRP DR

Operation: DRP

Description: Sets data register pointer to point to data register.

Example: DRP R25 Sets DRP to R25.

NOTE

The instructions to load DRP indirectly with RØ and to load ARP indirectly with RØ are:

DRP 1
ARP 1

Thus, to avoid confusion, R1 is not allowed in either the DR or AR fields. This means that CPU register R1 is for all practical purposes inaccessible except by means of a multi-byte RØ operation or when RØ = 1 and the ARP or DRP is specified by R*. See Using R* later in this section.

OTHER INSTRUCTIONS

In addition to the instructions above, there are a few other instructions which the programmer can use to manipulate quantities in the CPU and memory.

BCD

CPU Instruction

Set Decimal Mode

Format: DCM

Operation: DCM ← 1

Description: Sets DCM to 1 so that arithmetic operations will be in binary-coded decimal.

BIN

CPU Instruction

Set Binary Mode

Format: BIN

Operation: DCM ← 0

Description: Sets DCM to zero so arithmetic operations performed will be in binary.

CLE

CPU Instruction

Clear E

Format: CLE

Operation: E ← 0

Description: All four bits of the E (extend) register are cleared to zero.

DCE

CPU Instruction

Decrement E

Format: DCE

Operation: $E \leftarrow E - 1$

Description: E (extend) register decremented by 1. This instruction is always a binary operation, regardless of the setting of the DCM status flag.

ICE

CPU Instruction

Increment E

Format: ICE

Operation: $E \leftarrow E + 1$

Description: E (extend) register incremented by 1. This instruction is always a binary operation, regardless of the setting of the DCM status flag.

PAD

CPU Instruction

Pop ARP, DRP and Status

Format: PAD

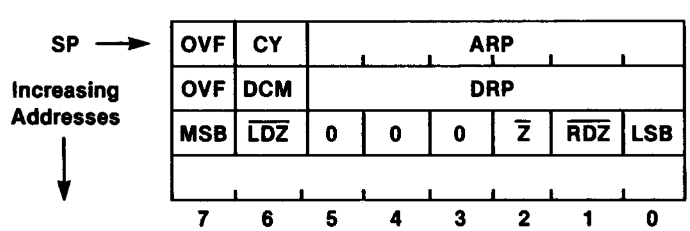
Operation: M(SP) → ARP, DRP and all status flags except E.

Description: Restore ARP, DRP and status (usually after a PAD instruction) by popping them off the stack.

Stack pointer is decremented by 3, and all status flags except E are altered by the contents of the three stack locations that are read.

The first byte processed is read as LSB in bit 0, $\overline{RDZ}$ in bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6 and MSB in bit 7. The second byte is read as DRP in bits 0-5, DCM status in bit 6, and overflow flags in bit 7. The third byte is read as ARP in bits 0-5, carry flag in bit 6, and overflow flag in bit 7.

Following a PAD instruction, the stack has been read as shown here:



RTN

CPU Instruction

Return From Subroutine

Format: RTN

Operation: SP ← SP — 2, PC ← M(SP)

Description: Subroutine return stack pointer is decremented by two. Then the return address is read from the stack and written into the program counter.

SAD

CPU Instruction

Save ARP, DRP and Status

Format: SAD

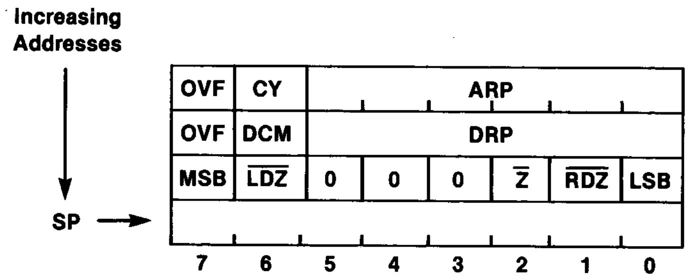
Operation: M(SP) ← ARP, and all status flags except E.

Description: Saves ARP, DRP and status (except E) in memory locations addressed by SP (stack pointer).

Three bytes are pushed onto the stack. The first byte contains ARP in bits 0-5, CY in bit 6, and the overflow flag in bit 7. The second byte contains DRP in bits 0-5, DCM status in bit 6, and the overflow flag in bit 7. The third byte contains LSB in bit 0, RDZ in bit 1, $\overline{Z}$ in bit 2, $\overline{LDZ}$ in bit 6, and MSB in bit 7.

SP is then incremented by three. Status is not affected by this operation.

Following a SAD instruction, the stack contents are as shown here:



USE OF R*

When entering source code, the programmer may substitute R* for the AR or DR in any CPU instruction. R* causes the ARP or DRP to be loaded with the least significant six bits of CPU register RØ. The effect is that the DR and AR are specified by the contents of RØ.

Example:	LDB RØ, = 26	Loads RØ with 26.
	LDB R*, R30	Loads CPU register specified by RØ (i.e., R26 now) with contents of R30.
	STB R40, R*	Stores contents of R40 into register (R26 now) specified by RØ.

ASSEMBLY OF CPU INSTRUCTIONS

When the address field of an instruction consists of a DR and an AR, each source statement is usually assembled into three bytes of machine code. These bytes are assembled in order as:

1. DRP: DRP set to point to DR.
2. ARP: ARP set to point to AR.
3. Opcode: Perform operation.

Thus, a stack push instruction such as PUBD would be assembled and appear as shown here:

Byte No.	Machine Code	Source Code
000227	110 006 342	PUBD R10, -R6

When the address field of an instruction consists of a DR and a label, as in the case of literal direct and literal indirect addressing (e.g., LDMI R32, = ADDRS), each source statement is usually assembled into four bytes of machine code:

1. DRP: DRP set to point to DR.
2. Opcode: Perform operation.
3. Low-order byte of literal quantity.
4. High-order byte of literal quantity.

When the address field of an instruction consists of DR, AR, and a label, as in the case of indexed direct and indexed indirect addressing (e.g., LDBI R36, X32, TABLE), five bytes of machine code may be generated for each source statement:

1. DRP: DRP set to point to DR.
2. ARP: ARP set to point to AR.
3. Opcode: Perform operation.
4. Low-order byte of address.
5. High-order byte of address.

HANDLING OF ARP AND DRP DURING ASSEMBLY

An optimizing feature of the Assembler ROM is the deletion of "unnecessary" ARP and DRP instructions during assembly.

If an instruction is not labeled (i.e., there is not an entry in the label field) and the ARP (and/or DRP) is already set to the correct value, the previously-set ARP (and/or DRP) is not generated during assembly.

For example:

Byte No.	Machine Code	Source Code
000227	110 006 342	LABEL POBD R10, -R6
000232	342	POBD R10, -R6

In this example, both the ARP and the DRP are specified beginning with byte 227. Since they are now correctly set for the next instruction, they are automatically deleted when the second POBD R10, -R6 instruction is assembled. This results in the machine code shown in byte 232.

Not all previously-set ARPs and DRPs are deleted during assembly. Instances where a previously-set ARP and/or DRP will not be deleted include:

--Labeled instructions. Since a jump from anyplace in code may cause execution to resume at the label, the first ARP and DRP are not deleted after an instruction that contains an entry in the label field.

--Returns. After executing a JSB, then returning, the first ARP and DRP encountered are not deleted.

--PAD. Following a PAD instruction, the first ARP and DRP are not deleted.

USING R#

When entering CPU instructions, the user may substitute R# in almost any instruction requiring an AR or DR. R# causes the ARP or DRP to be deleted from the machine code, regardless of other conditions. For example:

Byte No.	Machine Code	Source Code
000265	240	LABEL LDB R#, R#

R# is normally used after labels, when the ARP and DRP are already set correctly. By using R#, it is not necessary to squander time or bytes resetting ARP and DRP.

PSEUDO-INSTRUCTIONS

Pseudo-instructions are instructions to the assembler. Each may be entered by typing a pseudo-opcode in the same field as the opcode for an instruction, followed by any additional required operand.

Pseudo-instructions perform three main functions when encountered during assembly:

--Assembly control
--Data definition
--Conditional Assembly

PSEUDO-INSTRUCTIONS FOR ASSEMBLY CONTROL

ABS

Pseudo-Instruction

Absolute Program

Format: ABS 16
ABS 32
ABS ROM base address

Description: Declares an absolute program (i.e., with addresses that cannot be relocated), for either a computer with 16K bytes of memory, a computer with 32K bytes, or for a ROM beginning with the specified base address. If ABS 16 or ABS 32 is declared, the instruction must precede a NAM instruction.

FIN

Pseudo-Instruction

Finish Program

Format: FIN

Description: Signifies the end of the source code. This pseudo-instruction is required for assembly.

GLO

Pseudo-Instruction

Declare Global File

Format: GLO
GLO file name

Description: If no file name, declares this source code to be a global file. Otherwise, declares the global file to be used in the assembling of the current source code. Comments are not allowed on the same line as the GLO instruction, and the instruction must precede ABS and NAM.

LNK

Pseudo-Instruction

Link Files At Assembly

Format: LNK file name

Description: Will load another file containing more source code and continue assembling. Allows assembly of larger programs than would otherwise be possible.

Example: LNK SOURC2 When this instruction is encountered during assembly, the assembler looks for the file SOURC2 on the current mass storage device, loads the file, and continues assembling using the source code from the file.

LST

Pseudo-Instruction

List

Format: LST

Description: Causes the code to be listed on the current PRINTER IS device at assembly time. If the column width of the printer is sufficient (>46 characters) the listing will contain both the object and source code; otherwise, only the object code will be listed.

An address that is undefined when its label is encountered will be printed in object code as 326, 336, or 377, depending upon whether it is a DEF, a relative jump, or a GTO statement.

NAM

Pseudo-Instruction

Name Program

Format: NAM unquoted string

Description: Sets up the PCB (Program Control Block) for a binary program. Should be preceded only by GLO, ABS, LST, UNL, DAD, EQU, or comments. Illegal when ABS ROM has been declared.

Example: NAM KEYHIT Names a binary program KEYHIT and sets up the 32_8-byte program control block for that program.

ORG

Pseudo-Instruction

Origin

Format: ORG address

Description: Specifies a base address which is added to all following defined addresses (DAD's). This pseudo-instruction is most useful in global files.

UNL

Pseudo-Instruction

Unlist

Format: UNL

Description: Turns off the list feature which was turned on by the LST pseudo-instruction. After an UNL, code is not listed during assembly.

PSEUDO-INSTRUCTIONS FOR DATA DEFINITION

ASC
ASCII

Pseudo-Instruction

Format: ASC numeric value, unquoted string
ASC quoted string

Description: Inserts into the object code the ASCII code for the number of characters specified of the unquoted string. Inserts the entire quoted string.

Example:
ASC 3, FTOC   Inserts the ASCII code for FTO.
ASC 4, FTOC   Inserts the ASCII code for FTOC.
ASC "LOCATION"   Inserts the ASCII code for LOCATION.

ASP
ASCII With Parity

Pseudo-Instruction

Format: ASP numeric value, unquoted string
ASP quoted string

Description: Same as ASC except that the parity bit (MSB) of the string's final character is set. (During operation, the HP-83/85 system determines the end of an ASCII string in some system tables by checking to see if the character's parity bit is set. When the bit is found set, the system assumes the next character begins a new string or entry in the table.)

BSZ

Pseudo-Instruction

Bytes To Zero

Format: BSZ numeric value

Description: Inserts into the object code the octal number of bytes of zeros specified by the numeric value.

Example: BSZ 30 Fills 30_8 bytes with zeros.

BYT

Pseudo-Instruction

Bytes To Values

Format: BYT numeric value [,numeric value...]

Description: Inserts literal values into the object code.

Examples: BYT 377 Inserts octal 377 (i.e., all ones) into object code.

BYT 20,55C Inserts octal 20 into this byte of object code and BCD 55 into next byte.

DAD

Pseudo-Instruction

Direct Address

Format: Label DAD address

Description: Assigns either an absolute address or a constant to a label. DAD and EQU are similar; DAD is usually used for addresses, while EQU is used for values other than addresses. ORG affects only DAD's.

Example: INTORL DAD 56343 Assigns absolute address 56343 to the label INTORL.

DEF

Pseudo-Instruction

Define Label Address

Format: DEF label

Description: Inserts the two-byte address associated with the label.

Example: DEF RUNTIM Inserts two-byte address of the label RUNTIM.

EQU

Pseudo-Instruction

Equals

Format: Label EQU numeric value

Description: Assigns either an absolute address or a constant to a label. DAD and EQU are similar; DAD is usually used for addresses, while EQU is used for values other than addresses. ORG affects only DAD's.

GTO
Go To

Pseudo-Instruction

Format: GTO label

Description: Generates four bytes of object code which load the program counter (CPU registers 4 and 5) with the address minus one (i.e., ADR-1) of the label. The label must be for an absolute address.

The CPU relative jump instructions (JRZ, JNO, etc.) can cause jumps of from 177_8 to -200_8 memory locations. The GTO pseudo-instruction is useful for jumping beyond the range of relative jumps.

WARNING

The GTO pseudo-instruction is primarily for use in ROMs. It should not be used in a binary program unless that program has been declared an absolute program.

Example: GTO INTORL

VAL
Value

Pseudo-Instruction

Format: VAL label

Description: Inserts the one-byte literal octal value associated with the label.

Example: PPROM# EQU 360

VAL PPROM# Inserts the one-byte literal octal value (360) of the label PPROM# into the object code.

PSEUDO-INSTRUCTIONS FOR CONDITIONAL ASSEMBLY

This set of pseudo-instructions permits the user to control assembly by means of conditional assembly flags. A conditional assembly flag has the same constraints as a label--it can be no more than six characters in length, and the first character cannot be a digit.

A conditional assembly flag is treated the same as a label by the HP-83/85 system. (For example, an assembly flag can be located by a label search.) For this reason, a conditional assembly flag name should be unique, and should not duplicate a label.

AIF

Pseudo-Instruction

Assemble If Flag True

Format: AIF assembly flag name

Description: Tests the specified conditional assembly flag and, if true, continues to assemble the following code. If the flag tests false, the source code after the flag is treated as if it were a series of comments until an EIF instruction is found.

Example: AIF CYCLE Tests assembly flag CYCLE.

CLR

Pseudo-Instruction

Clear Flag

Format: CLR flag name

Description: Clears the specified conditional assembly flag to the false state.

Example: CLR CYCLE Clears assembly flag CYCLE.

EIF

Pseudo-Instruction

End Of Conditional Assembly

Format: EIF

Description: Terminates any conditional assembly in process. Only one conditional assembly can be handled at a time. If a second one is encountered while the first is still active, the second will override the first.

SET

Pseudo-Instruction

Set Flag

Format: SET flag name

Description: Sets the specified conditional assembly flag to the true state.

Example: SET CYCLE Sets conditional assembly flag CYCLE.

NOTES

SECTION 5

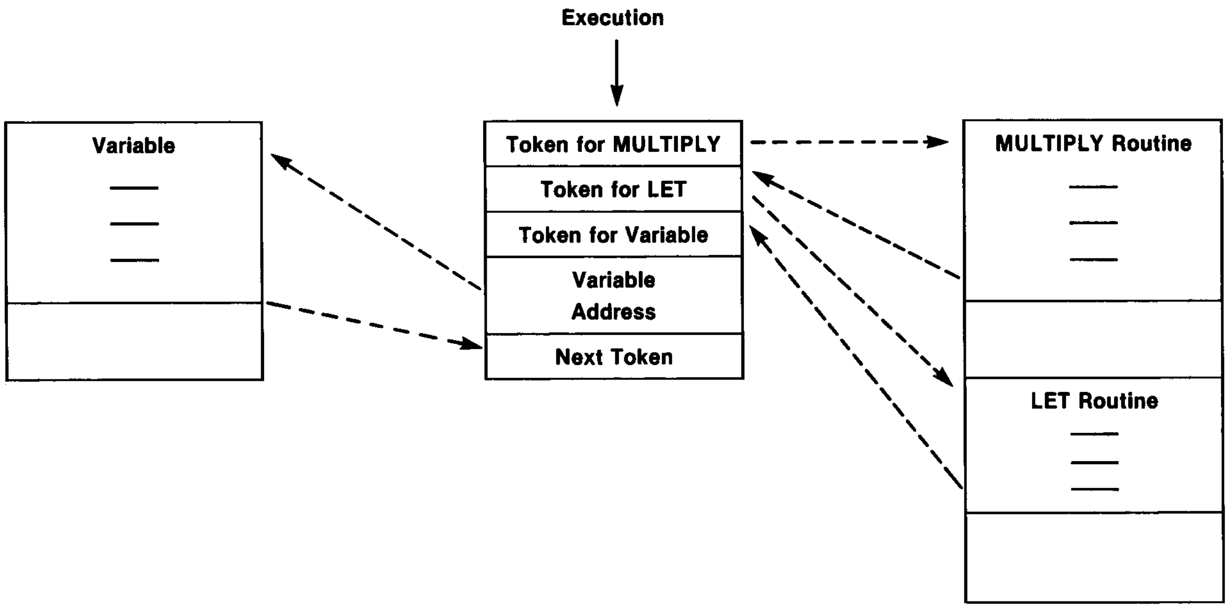
HP-83/85 SYSTEM ARCHITECTURE AND OPERATION

This section explains how system memory is allocated in the HP-83 and HP-85 computers, how programs are stored in that memory, and how a statement is parsed and becomes part of a BASIC program. It also explains the sequence of operations that occurs when a BASIC program is run.

In the computer, BASIC programs are executed by an interpreter that is part of the firmware operating system. However, the code that is interpreted is vastly different from the BASIC statements as they were originally entered. As the statements are entered, they are parsed and compiled into a form of RPN (reverse Polish notation), which can be interpreted more efficiently than the BASIC statements. As part of the parsing and compiling process, all BASIC reserved words are converted to single-byte tokens. This makes the internal form of the code somewhat more compact than the original form, and also makes interpretation easier and faster.

Also as part of the process of parsing and compiling, variables are placed in a variable storage area, with only their addresses remaining in the area containing the tokens.

A BASIC program, then, is held in memory as a series of tokens and addresses of variables. To execute the program, the computer processes these token and variable addresses in order. As each token is processed, it causes the machine to go to a table of routine addresses and execute a specific routine whose address is within that table. If the token indicates a variable, the machine uses the next two bytes as the variable address.

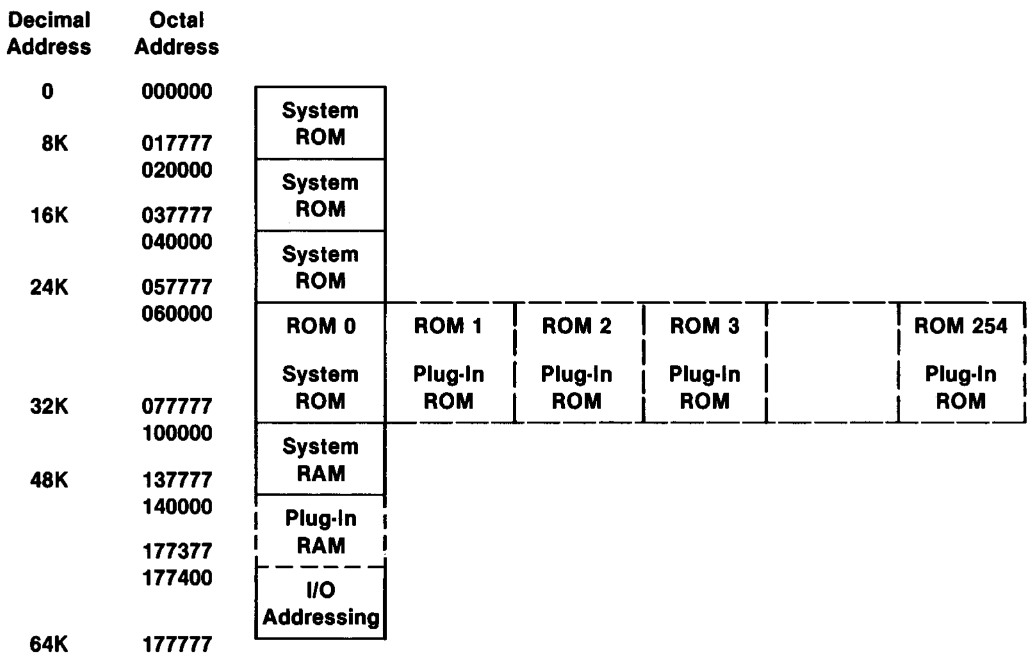


EXECUTION BY TOKENS

A binary program in memory, or a plug-in ROM, merely provides additional tokens (and their corresponding routines) to the set of HP-83/85 tokens and routines. This should become clear later in this manual.

SYSTEM MEMORY

The memory of the HP-83/85 is arranged as shown here:



SYSTEM MEMORY

As shown in the memory map, the main system contains three 8192_{10}-byte ROMs, the system ROMs. The fourth ROM space is for bank-selectable ROMs and it is shared by another system ROM and all plug-in ROMs. The only differences between the last system ROM and plug-in ROMs are that the select code for the system ROM is 0, and that the system ROM contains routines necessary for the HP-83/85 system to operate. Each plug-in ROM has its own unique select code. For example, the select code for the Assembler ROM is 40_{10}.

The last 256_{10} locations in the RAM address space are reserved for memory-mapping I/O addresses.

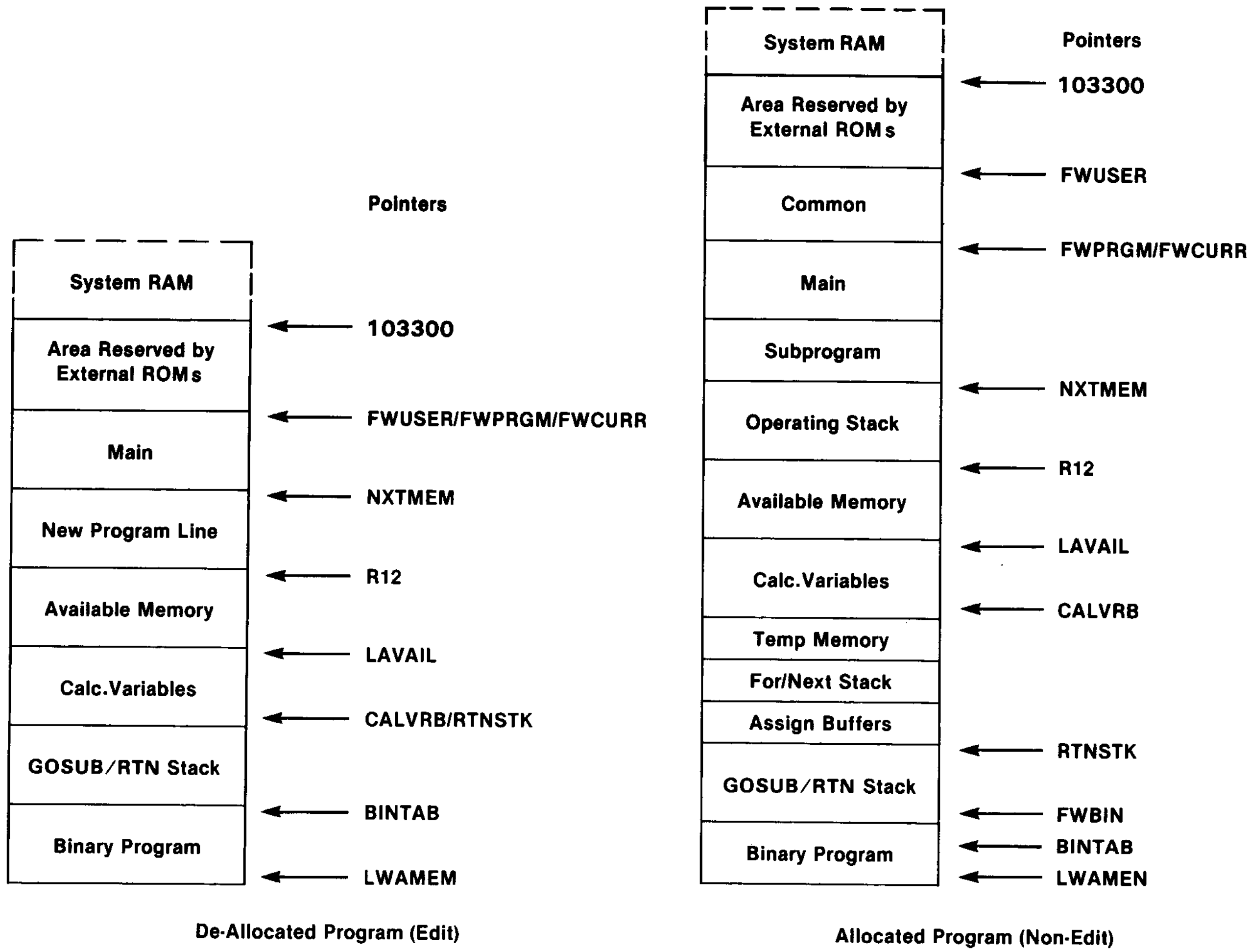
PROGRAMS IN MEMORY

There are two kinds of programs that can be resident in memory: BASIC programs and binary programs. In the HP-83/85, memory can contain a single BASIC program, BASIC subprograms, and a single binary program at one time. In addition, the computer can access the binary programs located in plug-in ROMs; these ROMs are bank-selectable by means of their select codes. In form and application, a plug-in ROM is closely akin to system ROM Ø or a binary program. Unlike a binary program, however, ROMs are not relocatable, and always begin with memory location 60000_8.

Within the HP-83/85, there are many pointers that are used to delineate and identify the different components of memory. Some of these pointers are in CPU registers, while others are at various locations in RAM.

ALLOCATION

A BASIC program may be resident in either allocated or de-allocated form. As a program is first entered from the keyboard, it is de-allocated and can still be edited. When a BASIC program is run for the first time, however, it must be allocated before it is actually executed. Memory that contains a de-allocated BASIC program appears as shown on the left below. An allocated program results in memory as shown on the right.



MEMORY AREAS

DE-ALLOCATED PROGRAM

When a BASIC statement is typed and [END LINE] is pressed, the computer checks for de-allocation. If the program is not already de-allocated, the HP-83/85 then de-allocates it.

In a de-allocated program, program variables are held as names rather than addresses, and the program can still be edited.

As illustrated above, in a de-allocated program the entire memory space is made up of RAM. The pointers that define the areas within RAM are:

FWUSER: FWUSER points to the first byte of RAM that can be accessed for a BASIC program by the user. FWPRGM points to the first byte of the main program. FWCURR is the first byte of the current program. These three pointers are all the same in a de-allocated program using the basic HP-83/85. (An external ROM that gives subprogram capabilities might cause these to be different.)

NXTMEM: NXTMEM points to the first byte after the end of the program as the program currently exists.

R12: CPU register R12 points to the execution stack. It is always used as an increasing stack, so R12 defines the first word of available program memory.

LAVAIL: This pointer defines the last word of available memory. LAVAIL actually points to the first word of the area where calculator variables are stored.

CALVRB and RTNSTK: These define the end of the calculator variables and the beginning of the BASIC subroutine return stack. These returns are the BASIC program's returns (and in a de-allocated program no returns exist here). These returns are not the same as those in a binary program, which are stored on the R6 stack.

BINTAB: Address of the first byte of the binary program. Although other pointers may change during allocation, BINTAB does not.

ALLOCATED PROGRAM

When a RUN, INIT, or STORE command is executed on the HP-83/85, the computer checks the allocation status of the resident BASIC program. If the program has not been allocated, the HP-83/85 allocates the program before executing further. Allocation creates variable space at the end of the BASIC program for all variables, and replaces all variable names with relative addresses. This allocation ultimately causes the program to be executed much more quickly.

The previous illustration of memory areas also shows an allocated program in memory. If common variables have been declared (that is, variables that are held in common by two BASIC programs), FWUSER points to the beginning of this common area, while FWPRGM points to the first word of the main BASIC program. (FWCURR points to the current program; this is the same as the main program unless an external ROM has provided subprogram capability.)

Such internal routines as print operations and string concatenation require temporary scratch-pad memory; this is provided as needed in the area directly after that addressed by CALVRB, and is released by the system immediately after the operation is performed. The FOR/NEXT stack is another temporary area that is provided when needed.

The Mass Storage ROM and the internal tape routines require 284_{10} bytes for each buffer (up to a maximum of 10 buffers), and these scratch-pad work areas are obtained in the buffer area directly above the GOSUB return stack.

SOFTWARE-DEDICATED CPU REGISTERS

Certain CPU registers are hardware-dedicated, and these registers always are used for the same tasks. Software-dedicated CPU registers are those registers which the system routines use for specific tasks. The registers and tasks vary, depending on whether the computer is parsing a statement, executing code at runtime, etc. However, here are the tasks of some of the most commonly-used CPU registers:

Execution Pointer: At runtime, registers R10 and R11 house the program counter (PCR), a pointer for executing a BASIC program. At parsetime, this pointer addresses the input stream.

Stack Pointer: Registers R12 and R13 contain the address of the operational stack pointer (SP).

Current Token: Register R14 contains the current token being processed in parse and decompile operations.

CSTAT: Register R16 contains CSTAT, which defines current status.

XCOM: CPU register R17 contains XCOM (external communication). The bits of this byte are used to discover why execution has halted, and to specify what to perform during the halt.

HP-83/85 OPERATION

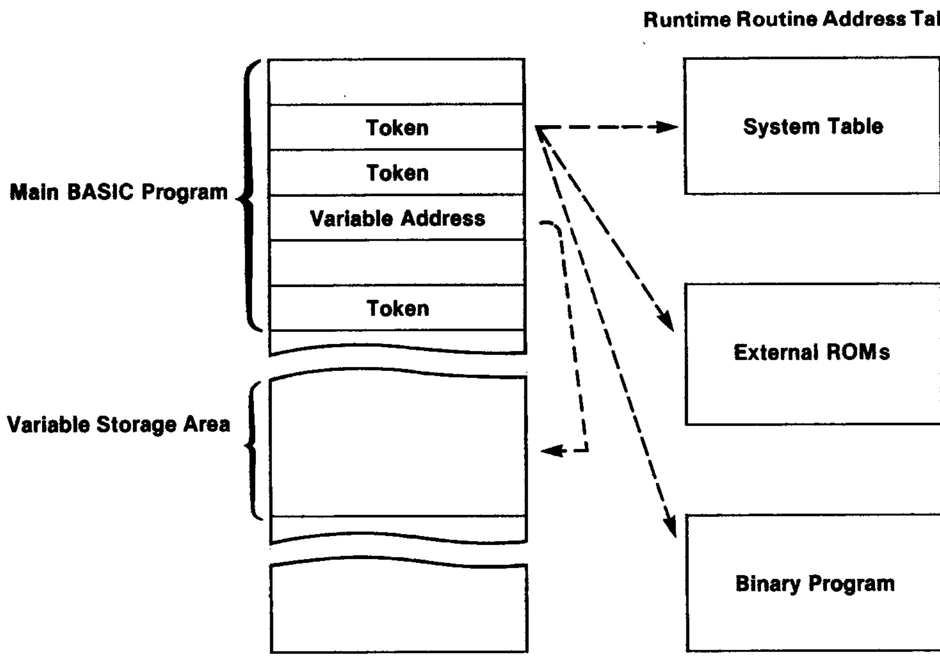
The basic HP-83 or HP-85 is controlled by system routines that are permanently resident at fixed addresses in memory. The addresses and names of many of these system routines may be found in the global file in section 7 of this manual.

In addition to the system routines, control can also pass to one of the plug-in bank-selectable ROMs, or to a binary program in the HP-83/85 memory. At certain times in the operation of the HP-83/85, the resident binary program and any ROMs are polled by the main system. In addition, there are a number of entry points, or "hooks," that allow HP-83/85 operation to be intercepted and modified by a binary program or ROM. These hooks normally do nothing in the system, but they can be used to take over the system at certain key times.

TOKENS

The HP-83 and HP-85 use tokens to represent the keyword, such as LET, FOR, BEEP, etc., that make up each BASIC statement. Each token is a one-byte quantity that indicates to the machine the addresses of routines associated with that token. Each token must have an associated entry in a table of routines for execution at runtime, another entry in an ASCII keyword table, and a third entry in a table of parse routines. A list of all system tokens may be found in appendix F.

The computer itself is a token-driven machine--a program is held in memory as a series of tokens and variable addresses, and the machine processes these tokens and addresses in order.



EXECUTION BY TOKENS (RUNTIME)

At runtime, for example, as the system executes a program, it processes a token by fetching the address of an associated runtime routine from a table of addresses. The runtime table may exist in a binary program and/or an external ROM as well as in the main system. The system performs a JSB to the specified address to execute the routine, then fetches the next token and searches for its runtime routine in the tables, etc.

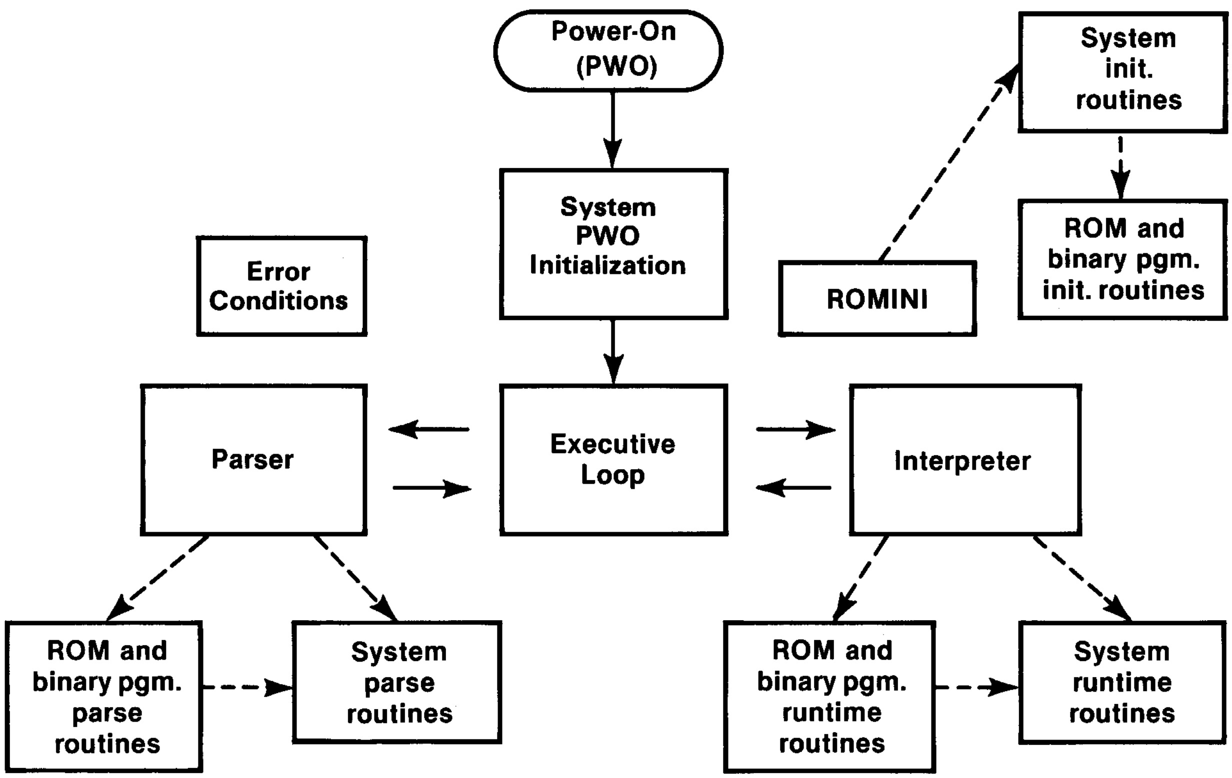
Some tokens indicate to the system that the two bytes following the token contain a variable address. In this case, the system processes the variable by locating it in one of the variable storage areas in memory.

Other tokens indicate that the bytes following the token are constants to be pushed onto the R12 stack.

Two tokens, 370_8 and 371_8, are used to expand the token tables. Token 370 indicates to the system that the next byte is the number of a ROM, and that the byte after the ROM# is the token within that ROM's tables that is actually to be executed. Token 371 directs the system to a binary program in the same way. More on these tokens later.

OVERALL SYSTEM FLOW

System flow in the HP-83/85 is shown by the flowchart below.



OVERALL SYSTEM FLOW

In general, loading and running a program, or executing a calculator mode statement, will require execution within the following areas:

Executive Loop: After power-on initialization, the system enters the executive loop and waits for some kind of action. The executive loop makes calls to the appropriate areas for initialization, parsing, allocation, running, and errors.

Parser: After a program line or calculator mode statement has been entered to the CRT, parsing occurs when [END LINE] is pressed. Parsing is the changing of ASCII code into tokens.

In parsing, the parser first searches the ASCII tables in the resident binary program for a keyword match, then searches the ASCII tables in any external ROMs, and finally searches the system tables.

Interpreter: The interpreter actually runs a program or executes a calculator mode statement by fetching tokens in order and calling the runtime routines to execute them.

In addition to the areas above, there are two other areas which may be called:

Initialization: At many times, including power-on, RESET, SCRATCH, etc., the system calls routines for initialization. Initialization routines are called through the ROMINI routine; the system polls system initialization routines first, ROM routines second, and the routine in the resident binary program last.

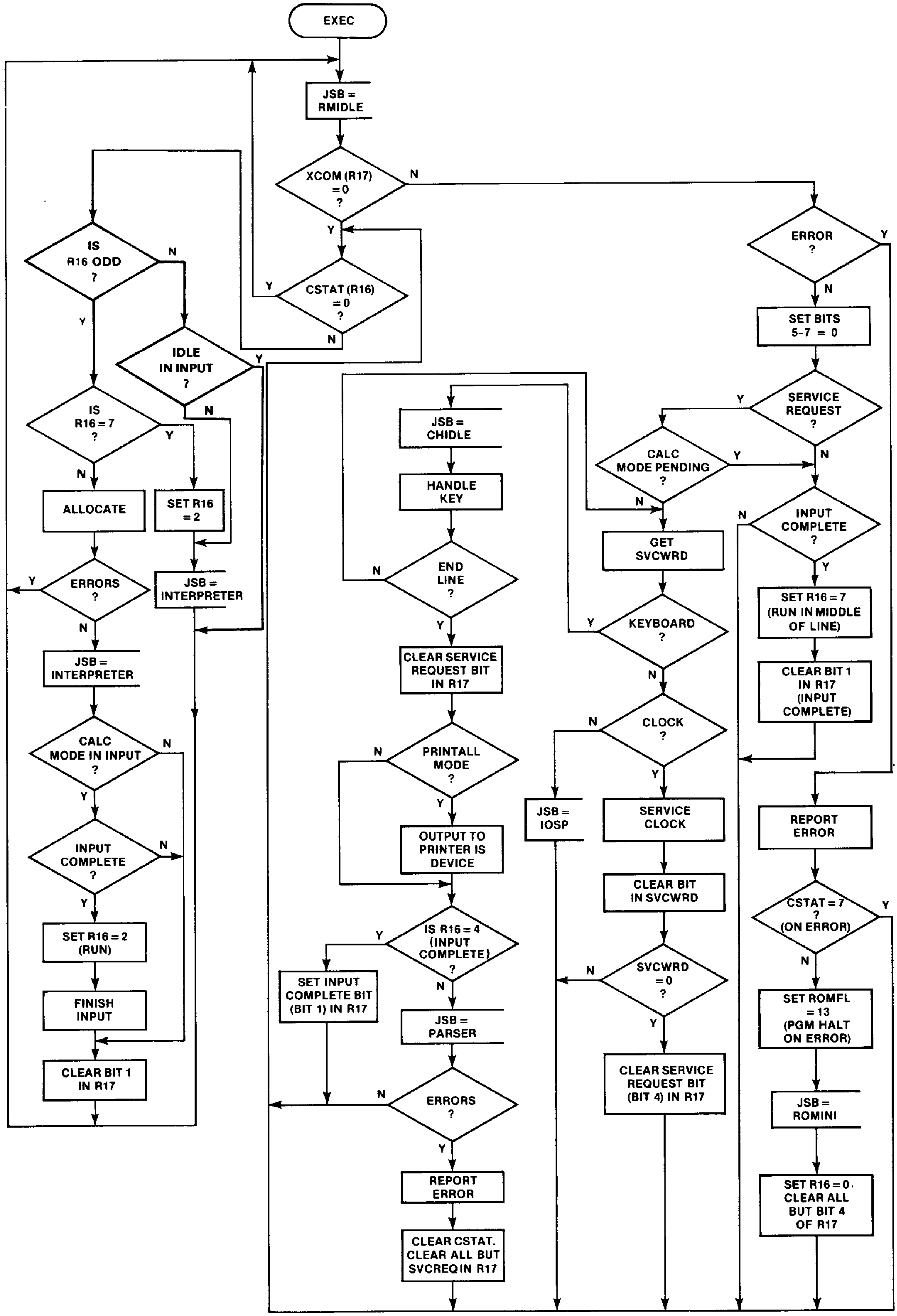
Errors: If errors are detected, the system generates the proper warning or error message.

EXECUTIVE LOOP

After power-on initialization, control passes to the executive portion of the system. The flowchart on the following page details the operation of this executive loop.

The executive loop itself contains a smaller loop that examines CPU registers R16 and R17 for status information. R16 contains CSTAT (current status), while R17 contains XCOM (external communication).

As long as the value of R16 is zero and all bits of R17 are set to zero, the system remains in the small loop. An interrupt, such as pressing a key, causes the system to leave the small loop and process the interrupt as shown on the flowchart.



EXECUTIVE LOOP

CSTAT

CPU register R16 contains an eight-bit word that is interpreted as current status.

CSTAT (R16) Value	Status
Ø	Idle.
1	Calculator mode.
2	Run mode. (Program is running.)
3	Not used.
4	Idle during input statement.
5	Calculating during input statement. (Evaluating expression before entering it as variable.)
6	Not used.
7	RUN in the middle of a line. (GOSUB or GOTO occurs because of a timer interrupt or soft key interrupt.)
8 - 255	Not used.

CURRENT STATUS

CSTAT is examined as an entire byte by the system.

XCOM

CPU register R17 contains XCOM, eight bits which are used for external communication of interpreter status.

XCOM (R17) Bits	Interpreter Halt
Bit Ø set	End of calculator mode.
Bit 1 set	Input complete.
Bit 2 set	Step mode.
Bit 3 set	Trace mode.
Bit 4 set	Service request. (Any interrupt sets this bit.)
Bit 5 set	Immediate set. (Can be set by user to halt interpreter.)
Bit 6 set	Error set.
Bit 7 set	Break. (OR of bits 5 and 6.)

INTERPRETER HALTS

During its cycles, the interpreter examines bit 7 of XCOM to determine if the interpreter is to halt. After an end-of-line token has been executed, the interpreter executive loop examines all bits of XCOM to see if control should be returned to the executive loop for further action. Any routine that sets bit 5 or bit 6 in R17 must also set bit 7, since the interpreter examines only bit 7.

HOOKS

Hooks into the executive loop are available through subroutine calls to RAM locations RMIDLE, CHIDLE and IOSP. In the normal system, each of these locations in RAM merely contains a return (RTN); they are present to allow the taking over of the executive loop by a binary program or external ROM.

ROMFL

ROMFL is a single-byte RAM location used to pass program conditions (such as RESET or RUN), to binary and ROM programs for initialization. Before the initialization routine in the binary program or external ROM is called, ROMFL is set to indicate the kind of condition that has occurred.

SVCWRD

SVCWRD is a RAM location that indicates the kind of interrupt.

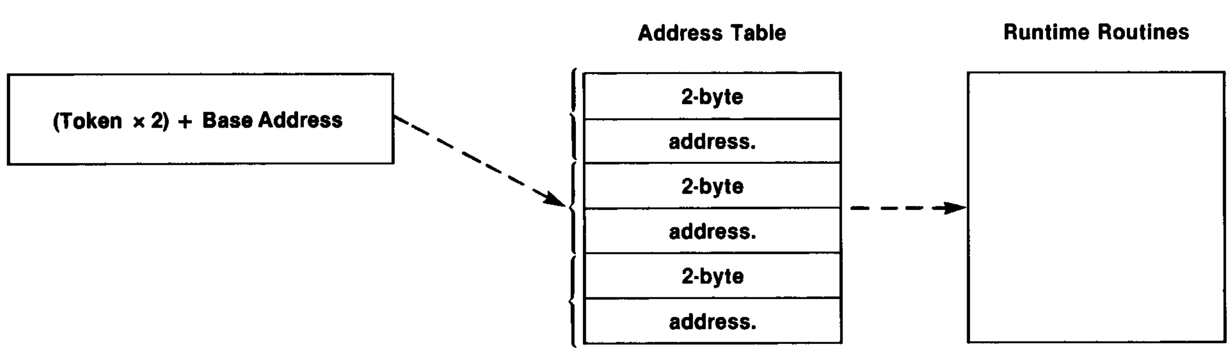
SVCWRD Bit	Type of Interrupt
Bit Ø set	Keyboard interrupt.
Bit 1 set	I/O interrupt.
Bit 2 set	Timer 1 interrupt.
Bit 3 set	Timer 2 interrupt.
Bit 4 set	Timer 3 interrupt.
Bit 5 set	Other interrupt.

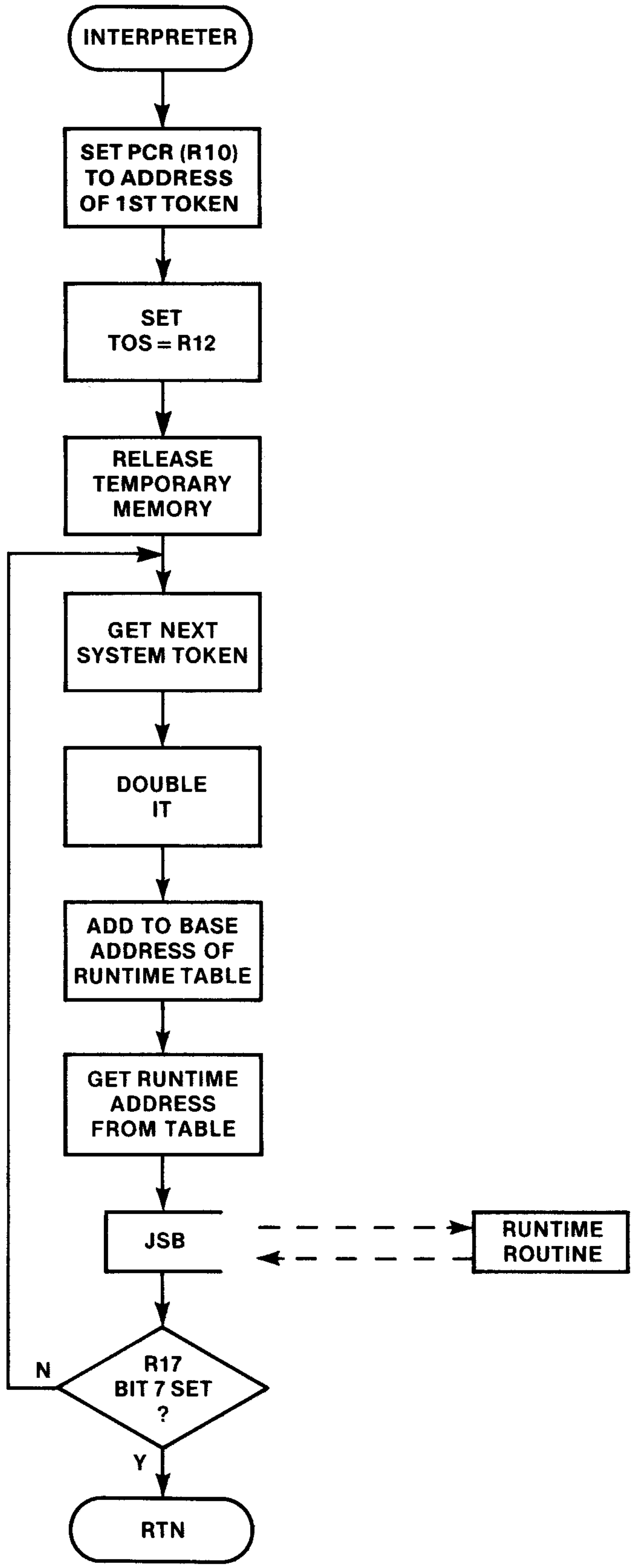
INTERRUPTS

INTERPRETER LOOP

The interpreter loop fetches the next token, processes it, and passes control to its runtime code. When the runtime code has been executed, control returns and the interpreter continues with another token. The following page shows a flowchart for the interpreter.

A token is an ordinal into a table of addresses. The address table is made up of two-byte addresses, so to find the actual address, the token is doubled, then added to the base address. This changes the ordinal into an offset pointing to the correct address.





INTERPRETER LOOP

After the runtime code is executed, the interpreter checks to see if the immediate break is set in R17, and processes the next token if it is not.

The procedure shown is for system tokens. Tokens 370 and 371 provide access to external tokens (that is, tokens whose tables reside in a binary program or ROM).

To find an external token, the interpreter first processes system token 370, doubles it, then adds it to the base address to find the system runtime routine for token 370. This runtime routine fetches the next two bytes via the R10 pointer; these bytes include the ROM number and the number of the token in the ROM or binary program. The runtime routine for the token 370 or 371 then handles this ROM or binary program token much the same way that the interpreter handles system tokens.

PARSING

As a line of a program or calculator mode statement is entered to the CRT, it is in ASCII code. When [END LINE] is pressed, the line is parsed. Parsing is the process of translating the ASCII code into the internal form in which programs are stored and run in the HP-83/85.

As a line is parsed, it is checked for syntax errors, changed to RPN (Reverse Polish Notation) from its original algebraic form, and converted into tokens that are then stored.

Each token consists of a single byte, and can represent a single keyword, such as LET or PRINT. Tokens 370 (ROM token) and 371 (binary program token) are used to allow extensions of the system by means of external ROMs and binary programs.

A table of system tokens may be found in appendix F of this manual. ASCII codes, which are also used during parsing, may be found in appendix E.

Parsing begins with the line number or the first character of the statement and moves to the right, processing each character and space. Multiple non-quoted spaces are compressed (i.e., ignored) during parsing.

Example: In parsing the line 10 LET A = B * SIN (45), the HP-83/85 system produces the following tokens in the order shown.

Tokens (Octal Values)	
20 0	Line number in BCD. (Two digits per byte.)
17	Length in bytes of statement.
142	LET token.
21	Store simple numeric variable token.
40 101	ASCII codes for the variable "blank A."
1	Fetch numeric variable token.
40 120	Blank B (in ASCII).
32	Integer token.
105 0 0	BCD 45 in integer format.
330	Sine token.
52	Multiply token.
10	Store numeric value token.
16	End of statement.

The stack addressed by CPU register R12 is used for parsing. A token is pushed onto the stack, the stack pointer is incremented, the next token pushed on, etc.

Parsing begins with the line number. This is loaded in BCD form; 20 is loaded first, since it is the least significant byte.

Next is the size or length of the statement. During parsing this is a blank placeholder byte; STSIZE is a pointer to the placeholder byte.

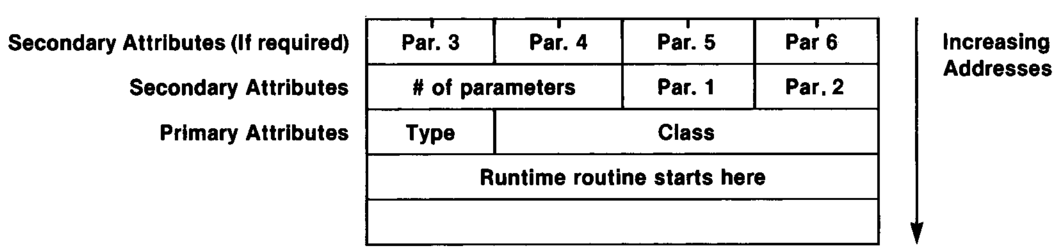
In order to find a match for the keyword LET, the system looks first in tables in the resident binary program, then in any external ROMs, and finally in the internal HP-83/85 system tables. (For this reason, a binary program or external ROM can take over any keyword.)

After parsing, if the statement was a program line, its tokens and addresses are inserted into the program space at the correct locations. If it was an expression or calculator mode statement, the parsed code remains on the R12 stack and is executed immediately.

For further details of parsing operations and register conventions at parse time, along with specific system parse routines, refer to Parsing in section 7.

ATTRIBUTES

In the process of parsing a BASIC statement, code is generated which consists of tokens and other information. For each token there is a set of attributes which define the type of token. The attributes occur immediately before the runtime code for the token.



ATTRIBUTES

Attributes are used to specify how parsing is to occur, how allocation and deallocation are to be performed, and how decompiling is to occur. They indicate to the system how the token is to be handled at these times. Attributes are not used at runtime.

There are two types of attributes: Primary and secondary. All tokens have primary attributes, but only BASIC language system-defined functions and operators have secondary attributes. The primary attributes immediately precede the runtime code. Secondary attributes occur before the primaries, and may occupy one or more bytes.

PRIMARY ATTRIBUTES

Within the primary attributes, the two most significant bits specify the token type. The next six bits specify the class.

TYPE

Bits 7 and 6 define the type of token.

Bits 7, 6	Type
11	BASIC statement, illegal after THEN.
10	BASIC statement, legal after THEN.
01	System command. (Non-programmable.)
00	Other (Not BASIC statement; e.g., function or operator.)

Examples:

Token	Primary Attribute	
DEF FN	3xx	Illegal after THEN.
LET	2xx	Legal after THEN.
DELETE	1xx	Not programmable.
SIN	0xx	Not a BASIC statement.

CLASS

The class indicates the form of the token. In many cases, the class is specific to a few tokens. A complete list of tokens and primary attributes may be found in appendix F, but the classes of tokens most often used in assembly language programming are shown here.

Class (Bits 5-0)	Token	Description
40		Immediate execute.
41		Other reserve words (i.e., most BASIC statements.)
42	100	Misc output (e.g., @ for special character handling).
44	31	Misc ignore. (Invisible at decompile time.)
50		Numeric unary operator. (e.g., –.)
51		Numeric binary operator. (e.g., +, –, *, /, \.)
52		String unary operator.
53		String binary operator. (e.g., &.)
55		Numeric system function (e.g., SIN, COS).
56		String system function. (e.g., CHR$, VAL$.)

USEFUL TOKEN CLASSES

SECONDARY ATTRIBUTES

Secondary attributes are used to specify the parameters for system-defined functions, as well as the precedence of numeric and string operators.

SECONDARY ATTRIBUTES FOR FUNCTIONS

A single byte can specify the parameter type for a function. A second byte is required only if there are more than two parameters. The first two bytes of secondary attributes are shown here.

7	6	5	4	3	2	1	0	Bit
Par. 3		Par. 4		Par. 5		Par. 6		Second Byte
# of parameters				Par. 1		Par. 2		First Byte

PARAMETER LOCATION

Parameter types must be included for all parameters used. The types are shown here.

Parameter Type	Description
0	Numeric
1	Numeric array
2	String
3	Not available

PARAMETER TYPES

SECONDARY ATTRIBUTES FOR OPERATORS

Secondary attributes also specify the precedence of numeric and string operators. The least significant four bits specify the precedence, as shown.

7	6	5	4	3	2	1	0	Bit
				P	P	P	P	

PRECEDENCE LOCATION

The precedence is defined within the HP-83/85 system as:

2	OR, EXOR
4	AND
6	Relational operators
7	+, -, Monadic +, monadic -, NOT
12	*, /,\ , DIV, MOD
14	^

(Some early versions of the HP-85 may have slightly different precedence.)

The only string operator is &, the concatenation operator, and it has a precedence of 7.

RUNTIME

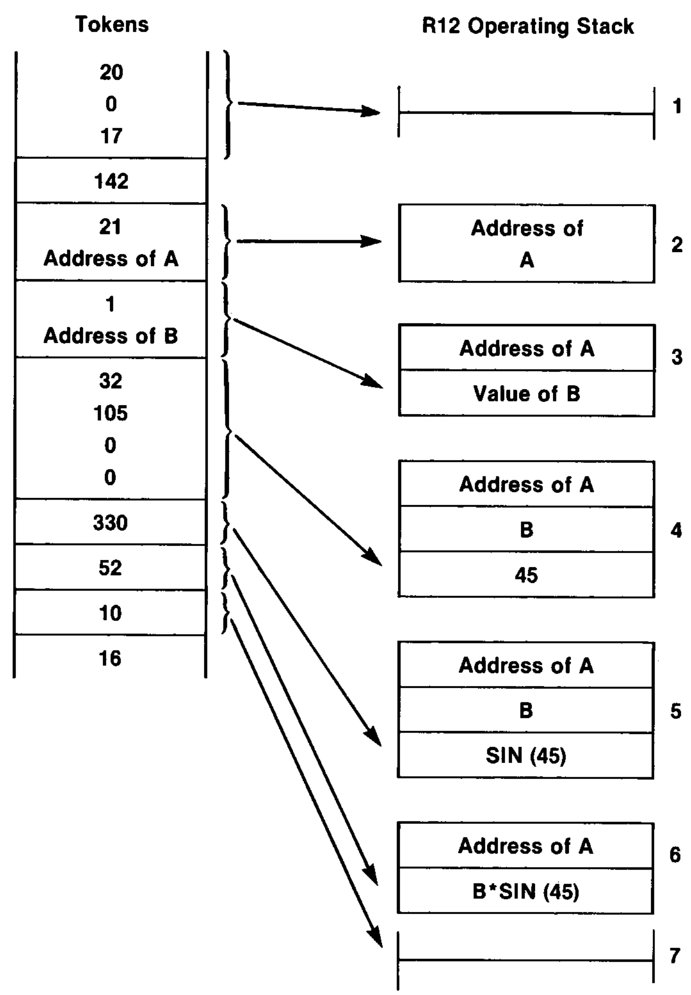
When a BASIC program is run, it is first allocated--all variable names are changed to relative addresses and all line references (such as GOTOs and GOSUBs) are changed to relative address references.

When the program is executed a token pointer (CPU register R10) is set to the first line of the main program, or to a specified line number, and control passes to the interpreter loop. The interpreter fetches a token, fetches the address of its runtime routine, and performs a JSB jump to the address to execute the routine there. The interpreter then fetches another token and execution continues to the end of the line.

Example: Recall the parsing for the line

10 LET A = B * SIN(45)

After parsing and allocation, tokens for the line are stacked in the program portion of memory as shown on the left below.



RUNTIME EXECUTION

R10 points to token 142. The interpreter passes over the line number (the first two bytes here) and the length (value 17, indicating that 17 bytes following belong in this line), then fetches token 142.

Token 142, the token for LET, is used as an index into the runtime table, a table of addresses which point to the runtime routines for the tokens. The interpreter fetches the address for the runtime routine for LET and causes a JSB to the routine. The LET routine does not affect the R12 stack.

After a return, the interpreter loop fetches the next token, and a JSB is done to that token's runtime routine. Since token 21 is the token for storing to a variable, the next two bytes (the variable address) are loaded from the token stream and pushed onto the R12 stack. These two bytes together give the address of variable A. The name block of variable A is also fetched from that address and pushed onto the R12 stack.

After a return to the interpreter loop, the runtime routine for the next token, 1, fetches a variable value. This fetch routine loads the next two bytes, which are the address of the variable from the token stream, and uses that address to fetch the value of variable B and push it onto the R12 stack.

After another return to the interpreter loop, token 32 causes the next three bytes to be loaded from the token stream and pushed onto R12 as an eight-byte tagged integer constant.

After a return, the next token, 330, causes a JSB to the sine routine. This routine expects a numeric value on the R12 stack; it calculates the sine of that value and pushes the computed result back onto R12.

The routine for the next token, 52, is the multiply routine. It expects two numbers on R12, and it pops these numbers off, multiplies them, and pushes the result back onto R12. The runtime routine for token 10 stores the value that is on the stack into the address of the variable that is on the stack.

Token 16, the end-of-line token, causes some internal clean-up (such as releasing any memory that might have been reserved by the line, etc.) and moves the run-time pointer past the line number of the next line to its first token.

For further details and specific system runtime routines, refer to Runtime in section 7.

DECOMPILING

Decompiling is the process of listing a program or statement. Internally, it requires the reconstruction of input code as it was entered to the CRT screen. The tokens which have been parsed into RPN and distributed in the system must be reassembled into algebraic notation. Decompiling is actually the reverse of the process of parsing and compiling.

Decompiling is a two-stack operation. An expression stack is used to reconstruct expressions from RPN to their original form, and an output stack is used to buffer the output. R12 is used for the expression stack.

In decompiling, the system processes each token and uses its class (a component of the token's primary attributes) to determine how the token is to be decompiled. Here are some common classes and how they are decompiled.

Class	Type of Token	Action
Ø	End-of-line	Unstack.
1	Fetch variable	To expression stack.
2	Integer	To expression stack.
3	Store variable	To expression stack.
4	Numeric constant	To expression stack.
5	String constant	To expression stack.
32	Subscript, e.g., A(3)	() to expression stack if token odd; otherwise (,) to expression stack.
34	Dimension subscript e.g., A$[ ]	[] to expression stack if token odd; otherwise [,] to expression stack.
36	Prints	Unstack and push , to output.
41	Other reserved words	If : then unstack, output reserved word, then unstack.
42	Miscellaneous output	If @ then push to expression stack and unstack; otherwise output.
44	Miscellaneous ignore	Ignore.
50	Unary operator	Insert after most recent missing operator in expression stack.
51	Binary operator	Replace most recent missing operator in expression stack.
52	String unary operator	Same as class 50.
53	String binary operator	Same as class 51.
55	System function	For each parameter, replace the most recent missing operator with , . Then insert function name and (at most recent missing operator and push) onto expression stack.
56	String system function	Same as class 55.

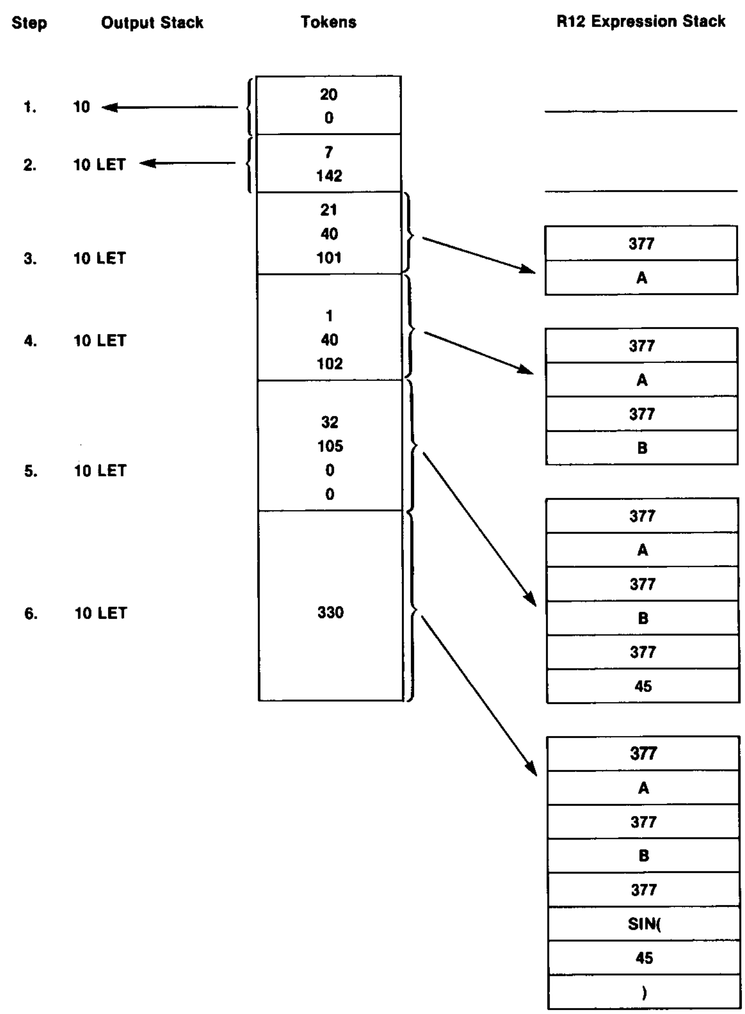
DECOMPILING BY CLASS

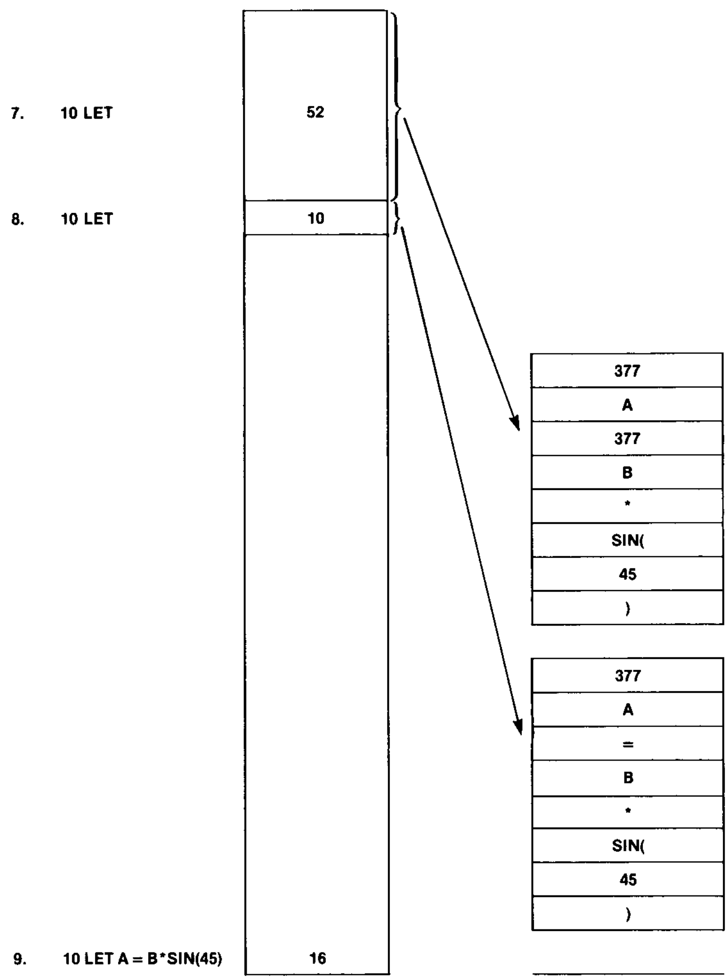
The following example should help illustrate how decompiling occurs:

Example: Recall again that the statement

10 LET A = B * SIN(45)

was parsed into the tokens shown below. These tokens are decompiled into the output stack and the expression stack as illustrated.





DECOMPILING

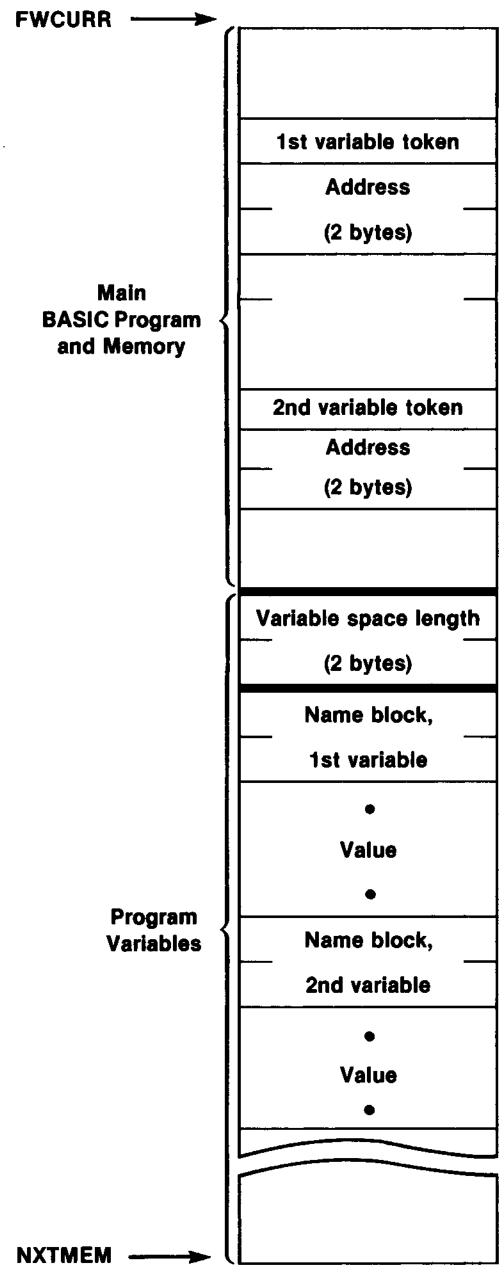
Since the tokens are arranged in RPN internally, as the system decompiles the tokens it pushes missing operator tokens (377) onto the expression stack. These missing operator tokens are merely "placeholders" until the arithmetic operators can be inserted at a later step.

Unlike parsing, decompiling is not an operation to which a binary program or ROM normally has access, since these programs are seldom required to perform any unique operations during decompiling. In some special cases the parse routines for a binary program or ROM may require modification if a statement is to be decompiled correctly. But for the most part, decompiling will not be a problem for the writer of binary or ROM programs.

For further details and specific system decompile routines, refer to section 7.

VARIABLE STORAGE

In the HP-83/85, variables may be stored in the variable storage area at the end of the BASIC program, in the common storage area, and in the area allotted for calculator variables.



VARIABLE STORAGE

In the main BASIC program, each variable is referenced by means of a token followed by a two-byte address. The variable itself is held in another part of memory, within the storage area for program variables. Immediately after the end of the BASIC program and available memory area in RAM is a two-byte quantity that signifies the beginning of variable storage and contains the length of the total space allotted for storage of that program's variables.

Each variable consists of a name block followed by the value of the variable. The two-byte variable address in the program is a relative one--it is actually a measure of the distance from FWCURR to the variable's name block in the storage area. The name block for each variable contains information about the variable. The format of the variable is shown here:

Byte	Bits 7	6	5	4	3	2	1	0
0	T3	T2	T1	T0	N3	N2	N1	N0
1	R2	R1	R0	L4	L3	L2	L1	L0

LEGEND

Bit	Meaning
T3	0 = Numeric
	1 = String
T2	0 = Simple
	1 = Array
T1, T0	0 = Real
	1 = Integer
	2 = Short
	3 = (Not used)
R2	0 = Local variable
	1 = Remote variable
R1	0 = Not being TRACEed
	1 = Being TRACEed
RØ	0 = Variable
	1 = Function value

N3 through NO and L4 through LO describe the variable name of the form A-Z or AO-Z9.

N3 through NO = Number minus 60_8; or 12_8 if blank.

L4 through LO = Alpha (ASCII) Code minus 100_8.

x In the following diagrams, x indicates the setting of the bit does not matter.

SIMPLE VARIABLE STORAGE

LOCAL VARIABLES

Byte								
0	0	0	T1	TO	N3	N2	N1	NO
1	0	R1	0	L4	L3	L2	L1	LO
2	Value							

8 bytes if real number.
4 bytes if short number.
3 bytes if integer number.

REMOTE VARIABLES

A remote variable is a common variable or a subprogram parameter passed by reference. Subprogram capabilities are available through some ROMs and these subprograms may have variables held in common.

Byte								
0	0	0	T1	TO	N3	N2	N1	NO
1	1	R1	0	L4	L3	L2	L1	LO
2	Pointer (2 bytes) to value							
3								

ARRAY VARIABLE STORAGE

LOCAL VARIABLES

Byte									
0	0	1	T1	T0	N3	N2	N1	N0	
1	0	R1	0	L4	L3	L2	L1	L0	
2	Total size as originally declared								
3	(2 bytes)								
4	Max row								
5	(2 bytes)								
6	Max column (377,377 if vector)								
7	(2 bytes)								
10	Row 0, column 0								
10+n	Row 0, column 1								n = Element size (3, 4, or 8)
10+n*m	Row 0, column m								m = Number of columns.
10+n*m+n etc.	Row 1, column 0								

REMOTE VARIABLES

Common area passed by reference.

Byte								
0	0	1	T1	T0	N3	N2	N1	N0
1	1	R1	0	L4	L3	L2	L1	L0
2	Pointer to total size							

STRING VARIABLE STORAGE

LOCAL VARIABLES

Byte								
0	1	0	x	x	N3	N2	N1	N0
1	0	R1	0	L4	L3	L2	L1	L0
2	Total length							
3	(2 bytes)							
4	Max length							
5	(2 bytes)							
6	Actual length							
7	(2 bytes)							
10+	String (as many bytes as required)							

Maximum length is the maximum number of characters that can be placed in the variable string. Actual length is number of characters currently in the variable string. Total length and maximum length are always the same unless:

--An I/O ROM is plugged in and this string is declared an I/O buffer.

--This string has been declared as a string array (using a ROM with advanced programming capabilities).

REMOTE VARIABLES

Common variable or subprogram parameter passed by reference.

Byte								
0	1	0	x	x	N3	N2	N1	N0
1	1	R1	0	L4	L3	L2	L1	L0
2	Pointer to total length							

FUNCTION STORAGE

The user-defined functions in a BASIC program (created with DEF FN) are stored in much the same manner as variables. Each is preceded in memory by a block that gives information about the function.

Because a function must restore status when it returns to a calling program, a stored function saves a return address (in R10), the BASIC program counter (PCR), the top-of-stack pointer (TOS), temporary memory, and calculator status (CSTAT).

In the illustrations below, the legend is the same as that for Variable Storage.

NUMERIC FUNCTIONS

Byte								
0	0	0	x	x	N3	N2	N1	N0
1	0	R2	1	L4	L3	L2	L1	L0
2	Function address							
3	(2 bytes)							
4	Return address							
5	(2 bytes)							
6	PC							
7	(2 bytes)							
10	TOS							
11	(2 bytes)							
12	CSTAT							
13	Numeric function value (8 bytes)							

STRING FUNCTIONS

Byte									
0	1	0	x	x	N3	N2	N1	N0	
1	0	R2	1	L4	L3	L2	L1	L0	
2	Function address								
3	(2 bytes)								
4	Return address								
5	(2 bytes)								
6	PCR								
7	(2 bytes)								
10	TOS								
11	(2 bytes)								
12	CSTAT								
13	Total length								
14	(2 bytes)								
15	Max length								
16	(2 bytes)								
17	Actual length								
20	(2 bytes)								
21	String function value								Number of bytes = total length. (Always 18 bytes.)

FORMATS ON THE R12 STACK

The stack to which CPU register R12 points is used for many operations by internal HP-83/85 system routines. The formats of variables that are fetched and stored during runtime execution of certain specific tokens, as well as the formats of numeric quantities, are shown below.

VARIABLES ON THE R12 STACK

The following table illustrates the format of variables on the R12 stack after the execution of certain tokens.

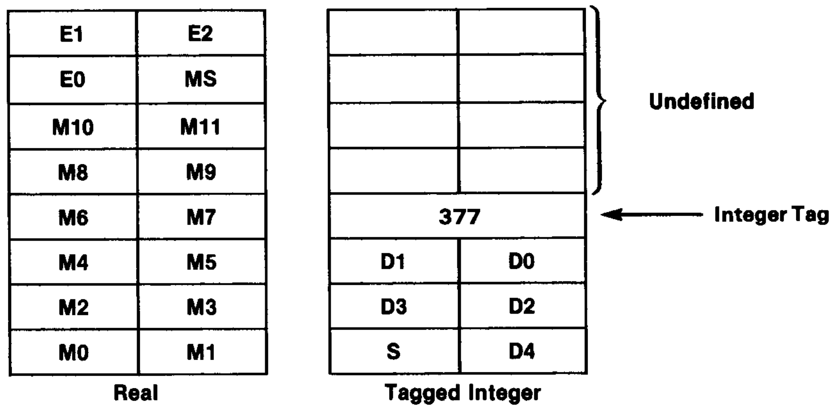
Token Executed	Places On R12 Stack	Number of Bytes
1	Value of simple variable.	8
2	Value of array element.	8
3	String length.	2
	String address.	2
21	Address of value storage area.	2
	Name block.	2
22	Absolute address of array variable area.	2
	Column. (Present only if TRACEing.)	2
	Row. (Present only if TRACEing, and array is two-dimensioned.)	2
	Dimension Flag. (Present only if TRACEing.)	1
	Name block.	2
23	Base address of string. (Relative if program mode, absolute if calculator mode.)	2
	Length available to store string characters in.	2
	Absolute address of 1st location available for storing characters.	2

When fetching or storing substrings, the address points to the first character of the substring.

Relative addresses are relative to FWCURR.

NUMERIC FORMATS ON THE R12 STACK

In internal HP-83/85 routines, all numbers popped off the R12 stack are eight bytes long, so integer values are tagged with octal 377.



NUMERIC FORMATS (R12 STACK)

In the illustration on the right, the byte above the number contains the octal quantity 377. This 377 acts as a tag for the number, specifying the quantity as an integer value that is only three bytes in length. The next four bytes popped off the stack are then undefined and are ignored by the system.

SECTION 6

WRITING BINARY AND ROM PROGRAMS

This section describes how to write a binary or ROM program. It outlines the parts of the program, and it also explains how a binary program or a ROM program is processed when it is assembled and when it is run.

Binary programs and ROMs are usually written to create new BASIC keywords or to take over and modify the operation of existing BASIC keywords.

There are almost no procedural differences in writing binary programs and ROMs. A binary program or a program for a ROM is written in an almost identical manner, using the HP-83 or HP-85, the Assembler ROM, and, if desired, the System Monitor. At assembly time, the object code for each is stored on a tape cartridge or disc. The object code for a binary program is then loaded back into the HP-83/85 to be run, while the object code for the ROM program may be read from the tape or disc into a commercial PROM/EPROM burner.

There are a few internal differences between binary programs and ROM programs. A binary program is usually relocatable, so that it may be loaded into computers with different sizes of memory. ROM program addresses must be absolute, but a ROM often needs to reserve some system RAM for its operation. Nevertheless, both binary programs and ROMs use the same set of HP-83/85 instructions and pseudo-instructions to generate source code.

Binary program and ROM source code is created using the instructions that make up the set of assembly language elements found in section 4 of this manual. These include the CPU instructions as well as the pseudo-instructions. The assembly language elements include, of course, subroutine jumps. These jumps can be used to actually call up internal HP-83/85 system routines for use in a binary program or ROM. It is often much easier to call a system routine to perform a function, rather than to painstakingly write the code to perform it. A list of available system routines and their addresses may be found in section 7 of this manual.

PROGRAM STRUCTURE

The structure, or "shell," of each binary program should be the same; this shell is shown below:

```
       NAM

       DEF  RUNTIM
       DEF  ASCIIS
       DEF  PARSE
       DEF  ERMSG
       DEF  INIT

PARSE  BYT Ø, Ø
       --Parse routine addresses go here.
RUNTIM BYT Ø, Ø
       --Runtime routine addresses go here.
       BYT 377, 377
ASCIIS BSZ Ø
       --Keyword table goes here.
       BYT 377
ERMSG  BSZ Ø
       --Error message table goes here.
       BYT 377
INIT   BSZ Ø
       --Initialization code goes here.
       RTN
       --The rest of the binary program goes here.
       FIN
```

<u>BINARY PROGRAM SHELL</u>

In order to examine the structure of a real binary program, look at the example program on the next page. The program creates a new BASIC statement, FTOC, for converting Fahrenheit temperature to Celsius. The function returns the Celsius equivalent of its Fahrenheit argument, according to the formula C = F-32*5/9. This program is one of the example programs on the Assembler Global File tape cartridge and disc.

Both the source code as it appears on the CRT and the object code are shown.

```
000000                  LST
000000          !***************************
000000          !*      FTOC BINARY        *
000000          !*  (c) Hewlett-Packard Co. *
000000          !*          1980           *
000000          !***************************
000000                  GLO GLOBAL
000000 106 124          NAM FTOC            !Creates program control block.
000002 117 103
000004 040 040
000006 002 000
000010 000 000
000012 000 000
000014 000 000
000016 000 000
000020 000 000
000022 000 000
000024 000 000
000026 000 000
000030 000 000
000032          !***************************
000032          !System Table:
000032 326 326          DEF RUNTIM
000034 326 326          DEF ASCIIS
000036 326 326          DEF PARSE
000040 326 326          DEF ERMSG
000042 326 326          DEF INIT
000044          !***************************
000044          !Parse Routine Table:
000044 000 000  PARSE   BYT 0,0
000046          !***************************
000046          !Runtime Routine Table:
000046 000 000  RUNTIM  BYT 0,0
000050 326 326          DEF FTOC.
000052 377 377          BYT 377,377
000054          !**************************
000054          !ASCII Table:
000054          ASCIIS  BSZ 0
000054 106 124          ASP "FTOC"
000056 117 303
000060 377              BYT 377
000061          !**************************
000061          !Error Message Table:
000061          ERMSG   BSZ 0
000061 377              BYT 377
000062          !**************************
000062          !Initialization Routine:
000062          INIT    BSZ 0
000062 236              RTN
000063          !**************************
000063          !Runtime Routines:
000063 020 055          BYT 20,55           !Attributes for FTOC.
000065          FTOC.   BSZ 0               !Begin runtime routine.
000065 230              BIN                 !Sets BIN mode for ONER routine.
000066 316 326          JSB =ONER           !Load F into R40.
000070 326
000071 150 040          LDM R50,R40         !Move F into R50.
000073 241
```

```
000074 140 251         LDM R40,=1,0,0,0,0,0,0,32C !Load 32 into R40.
000076 001 000
000100 000 000
000102 000 000
000104 000 062
000106 316 326         JSB =SUB10           !Perform subtraction.
000110 326
000111 170 012         POMD R70,-R12        !Throw away copy on stack.
000113 343
000114 150 251         LDM R50,=0,0,0,0,0,0,0,50C !Load 5 into R50.
000116 000 000
000120 000 000
000122 000 000
000124 000 120
000126 316 326         JSB =MPY10           !Perform multiplication.
000130 326
000131 170 012         POMD R70,-R12        !Throw away copy on stack.
000133 343
000134 150 040         LDM R50,R40          !Move intermediate result to R50.
000136 241
000137 140 251         LDM R40,=0,0,0,0,0,0,0,90C !Load 9 into R40.
000141 000 000
000143 000 000
000145 000 000
000147 000 220
000151 316 326         JSB =DIV10           !Perform division.
000153 326
000154 236             RTN                  !Answer is on stack, so return.
000155         ONER    DAD 56215
000155         SUB10   DAD 52137
000155         MPY10   DAD 52562
000155         DIV10   DAD 51644
                       FIN
```

The explanations on the following pages refer to this example program.

PROGRAM CONTROL BLOCK

The first 30_8 bytes of each BASIC and binary program are called a program control block. In the example program, the program control block appears in source code as:

```
10        LST
20        GLO GLOBAL
30        NAM FTOC
```

In a BASIC program, subprogram, or binary program these bytes contain information about the program. In a binary program, the following two bytes contain the absolute address at which the binary was last loaded. In the example program, this 32-byte section of code is reserved by the NAM statement.

A ROM does not contain this program control block. Instead, a ROM program is begun with the ROM number in the first byte and the ROM number complement in the second byte. A ROM program in memory will always begin at absolute location 60000.

In the example program, the NAM statement is preceded by the pseudo-op LST, which causes the object code to be listed during assembly of the program.

SYSTEM TABLE

Next in the example program is the system table for the program. This table is a list of addresses that in turn locate the runtime, ASCII, parse, and error message tables and the initialization routine farther down in the binary program.

```
60         DEF RUNTIM
70         DEF ASCIIS
80         DEF PARSE
90         DEF ERMSG
100         DEF INIT
```

The system table must always be present in a binary or ROM program, and it must always contain the addresses of subsequent tables in exactly the order shown here.

ROM Address		Contents	Binary Program Byte
60000 →	ROM#	Binary program base address ←	30
	ROM# complement		
60002 →		Address of runtime table ←	32
60004 →		Address of ASCII table ←	34
60006 →		Address of parse table ←	36
60010 →		Address of error message ←	40
60012 →		Address of initialization table ←	42

SYSTEM TABLE ADDRESSES

At certain times during operations such as initialization, parsing, running, keyboard entry, and error conditions, the HP-83/85 system expects an address of a table of addresses of routines for that operation to be in a specific location. If a binary program is resident during initialization, for example, the system expects in byte 42 of the binary program the address of an initialization routine. The system will use the contents of bytes 42 and 43 (whatever those contents are) for the address of the table.

PARSE ROUTINE TABLE

Next in memory is a table of addresses of the parse routines used by a program.

```
130 PARSE  BYT 0,0
```

In the example program there are no parse routines required. This is because the only keyword (FTOC) is a function, and thus has a syntax which can be understood and parsed by the HP-83/85. FTOC is a numeric function (attributes 20, 55) of one numeric parameter, just like SIN or COS.

The HP-83/85 automatically knows how to parse numeric and string functions because of their attributes. However, if a binary program or ROM creates a new BASIC statement, a parse routine will be required.

The data declaration pseudo-op BYT Ø, Ø is merely a filler to occupy the required opcode field. It corresponds to token Ø within the binary program. (Token Ø is illegal in the system and cannot be used.)

RUNTIME ROUTINE TABLE

The table of addresses that will be used during runtime follows.

```
160 RUNTIM BYT 0,0
170        DEF FTOC.
180        BYT 377,377
```

BYT Ø, Ø is again used as a filler. When executing object code, the system locates the address for RUNTIM, skips two bytes, then uses the next two bytes as an address for the first runtime routine.

A common convention, although not one that is required, is to name runtime routines with the keyword (or an abbreviation) followed by a period.

The pseudo-instruction BYT 377, 377 inserts two bytes with all bits set. This signifies the end of all addresses to be relocated during loading of the binary program.

The address tables for binary programs are relative when assembled. When the LOADBIN instruction is executed, the object code is first loaded, then some relocation is done. All pointers up to the first occurrence of an octal 377, 377 are adjusted. This is necessary because the ASSEMBLE command stores a program without readjusting the pointers and because the machine into which the program is later loaded may not have the same memory size as the one which stored the code.

Since a ROM program is not relocatable, the 377, 377 "marker" is not required in a ROM.

ASCII TABLE

The next component of the program is the table that contains the ASCII keywords.

```
210 ASCIIS BSZ 0
220        ASP "FTOC"
230        BYT 377
```

In an ASCII table, all of the keywords are arranged sequentially. When a BASIC statement is entered to the CRT, the system attempts to match the characters that are entered with a keyword in one of the ASCII tables. It looks first in the resident binary program, then in any plug-in ROMs, and finally in its own ASCII tables for a match.

The system attempts to find a match, processing a character at a time until it reaches a character with its most significant bit set. A character with its MSB set signifies the last character of a keyword. If no match has been found, the system assumes the next character in the tables begins a new keyword, and it moves to that character, increments a token counter, and begins trying once again to find a match.

In the example program, the ASP pseudo-instruction causes the most significant bit of the C in the keyword "FTOC" to be set. BYT 377 sets all the bits in one byte, signifying the end of the ASCII table.

ERROR MESSAGE TABLE

Like the other tables, the address of the error messages is required in a binary program.

```
260 ERMSG   BSZ 0
270         BYT 377
```

In the example program, there are no error messages. Errors during parsing will be reported by the system, since system routines are performing all parsing; and runtime errors will be trapped by the math routines used. Again, BYT 377 signifies the end of the table.

INITIALIZATION TABLE

This section of the program contains the address of a routine that is executed during initialization. This section is entered during power-on, reset, allocation, deallocation, and at other times. The flag in memory location ROMFL indicates which of these entry possibilities has occurred.

```
300 INIT    BSZ 0
310         RTN
```

The example program does not require any specific action during initialization, so all that is required is a return. For an example of ROMFL usage, see the Special Function Keys as Typing Aids example program in section 8.

RUNTIME ROUTINES

This section contains most of the code used in the program, and normally includes many runtime routines. Routines here must be included in the tables above; otherwise, the system will not be able to access these routines. In the example program, there is only a single runtime routine mentioned in the tables above: "FTOC."

During parsing, when the system finds the routine address for a particular keyword (FTOC., in this case), it examines the primary attributes, located one byte before the runtime code. (It also examines secondary attributes, if required.) The attributes define for the system the type of keyword--statement, function, operator, etc.--so that the system can process the keyword properly.

The attributes 20, 55 specify that the next keyword, FTOC., is a numeric function with one numeric parameter, so the system knows how to parse a statement that contains the keyword FTOC, and it knows how many parameters to accept at runtime.

Next is the runtime code for FTOC. The calculation to be performed is C = (F-32) *5/9; the FTOC routine takes an argument off the R12 stack, subtracts 32 from it, multiplies the result by 5 and divides that result by 9. Like all functions, FTOC leaves the final result on the stack.

```
340          BYT 20,55              !Attributes for FTOC.
350 FTOC.    BSZ 0                  !Begin runtime routine.
360          BIN                    !Sets binary mode for entry to ONER routine.
370          JSB =ONER              !Load F into R40.
380          LDM R50,R40            !Move F into R50.
390          LDM R40,=1,0,0,0,0,0,0,32C !Load 32 into R40.
400          JSB =SUB10             !Perform subtraction.
410          POMD R70,-R12          !Throw away copy on stack.
420          LDM R50,=0,0,0,0,0,0,0,50C !Load 5 into R50.
430          JSB =MPY10             !Perform multiplication.
440          POMD R70,-R12          !Throw away copy on stack.
450          LDM R50,R40            !Move intermediate result to R50.
460          LDM R40,=0,0,0,0,0,0,0,90C !Load 9 into R40.
470          JSB =DIV10             !Perform division.
480          RTN                    !Answer is on stack, so return.
```

Refer to section 4 for descriptions of the CPU instructions and pseudo-instructions used. Refer to section 7 for descriptions of the system routines (such as ONER and MPY10) used.

EXTERNAL LABEL TABLE

After the runtime routine is a label table. The label table gives the addresses in RAM of the system routines used in the binary program. Unlike the binary program's own routines, there are no addresses available for system routines unless the addresses are specified in some manner. These addresses will be found in the system global file (listed in section 7 of this manual) and/or in the listings of individual system routines in the same section. In the example program, the table of system label addresses is placed at the end for easy reference, but it can be placed anywhere in the program after the BYT 377, 377 marker.

```
490 ONER    DAD 56215
500 SUB10   DAD 52137
510 MPY10   DAD 52562
520 DIV10   DAD 51644
```

If the addresses for all system routines used in a program are available on a global file on disc or tape (such as the Assembler Global File), a label table need not be written. Instead, the program can be directed to look in the system global file by means of the GLO pseudo-instruction. Merely place a GLO GLOBAL instruction before the NAM instruction and ensure that the source file named GLOBAL is available on the tape or disc when the program is assembled.

The user may also create a global file by assembling a list of DAD's and EQU's, with GLO as the first statement.

ENDING THE PROGRAM

FIN is used to terminate assembly; LNK is used to cause assembly to resume with another section of source code.

```
530         FIN
```

SYSTEM HOOKS

The main reason for an external ROM or binary program is to extend the capabilities of the main system. In order to allow for this, a number of hooks are provided. A hook is a location where a binary program or ROM can gain control of the system. There are three main categories of hooks: Language hooks, general hooks, and initialization hooks.

LANGUAGE HOOKS

With language hooks the binary program or ROM can define new keywords, functions, and auxiliary tokens. Because the system first polls the resident binary program, then all external ROMs, and finally its own system tables when searching for these, a binary or ROM program can take over or supersede any of them.

GENERAL HOOKS

To provide for each general hook, the system at certain times executes a JSB subroutine jump to a specific RAM location. During normal operation each of these RAM locations contains a RTN (return) or is otherwise idle. By placing a JSB to a binary program or ROM at the hook location, the program or ROM gains access to the operating system. It is the responsibility of the writer of the external program to determine how to use the hook and how to avoid conflict with other usages of the hook. No support is supplied by the system.

Unless otherwise noted, each general hook is seven bytes in length. General hooks are supplied at the following points:

RAM Name	Location	Function
IOTRFC	102400	I/O Traffic intercept. Used by I/O and P/P ROMs.
IOSP	102407	I/O Service pointer. Used by I/O and Mass Storage ROMs.
CHIDLE	102416	Character editor intercept.
KYIDLE	102425	Keyboard intercept. Polled whenever a key is pressed.
RMIDLE	102434	Executive loop intercept.
IMERR	102452	Image statement errors. Located in image code. Used by I/O ROM.
PRSIDL	102461	Parser intercept.
IRQ20	102470	I/O Interrupt (9 bytes). Interrupt vector, like keyboard service and clock routines.
SPAR0	102501	Spare interrupt (9 bytes). Hardware interrupt vector hook. Used by System Monitor.
SPAR1	102512	Spare interrupt (9 bytes). Hardware interrupt vector hook.

GENERAL HOOKS

At power-on, the first two general hooks above are initialized to JSB = ERROR+, BYT 25. The remaining eight are initialized to RTN.

The following section of code illustrates how to take over a hook (in this case, the CHIDLE hook):

```
LDM R36, = KEYCHK        Load address of routine to handle CHIDLE.
ADMD R36, = BINTAB       Add value of BINTAB for an absolute address.
STM R36, R45             Store desired address in R45 and R46.
LDB R47, = 236           Load the opcode for return (RTN).
LDB R44, = 316           Load the opcode for JSB.
STMD R44, = CHIDLE       Store it all (multi-byte store) to CHIDLE hook.
```

INITIALIZATION HOOKS

A routine called ROMINI is called on several occasions to perform initialization in external programs. When this occurs, the initialization routines in binary program and ROMs are given control.

A parameter is passed to the ROMINI routine by way of ROMFL, a single-byte RAM cell. The occasions and corresponding ROMFL are:

ROMFL Value	Function
0	Power on
1	RESET key
2	SCRATCH
3	LOADBIN
4	RUN, INIT
5	LOAD
6	STOP, PAUSE
7	CHAIN
10	Allocate token with class > 56
11	Deallocate token with class > 56
12	Decompile token with class > 56
13	Program halt on error

These calls to the ROMs and binary program allow these programs to initialize, de-initialize, and otherwise keep track of operation. For instance, if a ROM needs to reserve or "steal" memory permanently, it would check for ROMFL = Ø, and reserve memory only when that is true. Another example is that during RESET the I/O ROM might want to deallocate buffers.

During initialization, a binary program or ROM should never destroy any CPU registers below R20. Similarly, no initialization routine should use CPU registers other than R34-R37 until it is verified that the value of ROMFL is not 10, 11, or 12. Once the value of ROMFL is not 10, 11, or 12, all CPU registers numbered R20 or higher may be used.

ERROR MESSAGES

ROMs and binary programs have the option of reporting system (predefined) errors or reporting their own error messages. System and ROM errors use positive error numbers, while error messages defined by a binary program are referred to by negative error numbers.

USING SYSTEM ERROR MESSAGES

HP-83/85 system errors can be used in binary programs and ROMs in the same way they are used for system programs. This involves a subroutine jump to system routine ERROR or ERROR+, which expect the next byte to contain the desired error number.

Example:

```
JSB = ERROR                         Set errors.
BYT 37                              System error 37.
Anything                            Continuation after error.
```

Example:

```
JSB = ERROR+                        Set errors and return.
BYT 37                              System error 37.
```

No return is necessary. ERROR+ throws away one return address before performing a RTN.

This last section of code is equivalent to:

```
JSB = ERROR
BYT 37
RTN
```

ROM-DEFINED ERROR MESSAGES

When setting up an error message table for a ROM, remember that the first eight error messages are warnings; they should have default conditions such as in the ROM error message table shown here:

```
ERMSG  BYT 200, 200, 200, 200 }       Eight dummy bytes with
       BYT 200, 200, 200, 200 }       MSB set.
       ASP "SYSTEM DOWN"              Error #11(8).
       ASP "BAD INPUTS"               Error #12(8).
       ASP "WALK AWAY"                Error #13(8).
       BYT 377                        End of error message table.
```

Error messages defined in a specific ROM can be selected by first storing the ROM number in a location known as ERRROM, then calling system routine ERROR or ERROR+. Since it is possible for multiple errors to occur before they are reported, location ERRORS contains a flag that signals whether any errors have already occurred; once ERRORS is set, ERROR throws away all subsequent errors.

Here is a section of code that would be located within a ROM to check for any prior errors, then load ERRROM with the ROM number for error reporting:

```
ERRSET LDBD R36, = ERRORS             Get error flag.
       JNZ DON'T                      Jump if already an error.
       LDB R36, = 40D     }           Otherwise load ROM number
       STBD R36, = ERRROM }           (40(10) in this case) into ERRROM.
DON'T  RTN
```

To report errors within ROM #50, the reporting code would first call the above routine, then call ERROR or ERROR+, as shown in this example:

LDM R26, R36	
SBM R26, R24	
JZR GOAHED	
JSB = ERRSET	Select proper ROM number.
JSB = ERROR+	Report error 12. ("BAD INPUT"
BYT 12	in earlier error message table.)

Note that ERROR or ERROR+ will do nothing if ERRORS is already set, so no testing is required after calling ERRSET.

BINARY PROGRAM ERROR MESSAGES

As in a ROM, the first eight errors within a binary program are warnings and should have default conditions. Unlike system or ROM errors, however, binary program errors are referenced by negative error numbers. Here is an example of a binary program error message table:

ERMSG	BYT 200, 200, 200, 200	Eight dummy bytes (377-370)
	BYT 200, 200, 200, 200	with MSB set.
	ASP "BAD PARAMETER"	Error $\#367_8$.
	ASP "WILD CARD PROBLEM"	Error $\#366_8$.
	ASP "INPUTS LOST"	Error $\#365_8$.
	BYT 377	End of error message table.

When the correct error is found, the error number is reported in two's complement form. The following section of code illustrates how an error message from the binary program error message table might be called:

POMD 22, -12	Get a number.
JNZ OK	Jump if not zero.
JSB = ERROR+	Otherwise, report error
BYT 367	#367, "BAD PARAMETER."

BINARY PROGRAM AND ROM ADDRESSING

Functionally there is no difference between a binary program and an external ROM; any task which can be performed by one can be done by the other. Each has special problems, however, related mostly to addressing.

EXTERNAL ROM ADDRESSING

External ROMs are selectable by software, so a special problem occurs when selecting among ROMs.

Suppose it is desired that an external ROM call the TIME function. This function is located at address 65517 in the bank-selectable system ROM (i.e., ROM Ø).

Because the external ROM occupies the same address space, it is impossible to directly select system ROM Ø, execute a JSB to the TIME routine, and return to the calling ROM.

The solution is to call the system routine to be executed (TIME) through a system routine called ROMJSB. Two parameters are passed to ROMJSB:

1. Address of the routine to be called.
2. ROM number of the location where the routine resides.

Example: To call the TIME routine, the source code in the external ROM would be:

```
JSB = ROMJSB                      Call to ROMJSB.
DEF TIME.                         Address of routine to be called (TIME).
BYT Ø                             Number of ROM that contains TIME.
```

When the TIME routine has been executed, control returns to the ROMJSB routine. ROMJSB, in turn, reselects the calling ROM and returns execution to the next instruction after BYT Ø.

Another problem is how to return to the system ROM. It is impossible to select ROM Ø and then return, because selecting ROM Ø deselects the ROM which is trying to execute a return. The solution is another system routine called ROMRTN, which performs the same function (select ROM Ø and return). In most cases the system automatically reselects ROM Ø after a normal return, but in some cases, such as after all parse routines, the external ROM must "clean up" by selecting ROM Ø before returning. Executing GTO ROMRTN reselects ROM Ø and then returns.

A third problem is the overhead required to intercept a system routine. Several general hooks have been provided; for example, in the executive loop a subroutine jump is made to a RAM location (a system hook) called RMIDLE. At power-on, the system stores a RTN at that location. To intercept the idle loop, a ROM must load the following sequence into that location (and the following six bytes).

```
RMIDLE JSB = ROMJSB                     Call ROM switching routine.
       DEF INTERC                       Address of routine to be executed.
       BYT 17                           ROM number.
       RTN                              Return.
```

The load can be performed by the ROM's initialization routine when the ROM gains control during power-on initialization (ROMFL = 0).

For a binary program to take over the same hook, all that is needed is:

```
RMIDLE JSB = INTERC
       RTN
```

One further general caution is that any routine which calls an external ROM, such as an interrupt service routine, must also use the ROMJSB utility. This is true even if the external ROM is called from a binary program.

BINARY PROGRAM ADDRESSING

The addressing problem of binary programs is relocatability. The HP-83/85 processor accomodates relocatable code. All conditional jumps and the JMP command are relative, so they are inherently relocatable. Arithmetic, loads, stores, and subroutine jumps can all be performed in an indexed mode. If a two-byte register contains a base address stored in RAM, such as BINTAB, then relocatable code can be written using indexed addresses (indexed by the base address).

Examples of the various operations follow. The examples assume CPU registers R36 and R37 contain the base address of the binary program. The base address will be stored in BINTAB (101233) by the system LOADBIN command.

Examples:

```
       LDMD R36, = BINTAB           Load up base address.
       JSB X36, DEST.               JSB to destination DEST.
       LDM R40, X36 CONST           Load a constant into R40.
       LDMD R22, X36, ADDR          Load direct R22.
CONST  BYT 12, 34, 56, 70
       12, 34, 56, 70
DEST.  RTN                          Short subroutine.
ADDR   BSZ 2                        Address in main memory.
       FIN                          End of program.
```

All of the labels in this section of code are merely examples.

RESERVING RAM

A binary program or ROM sometimes requires that system RAM be "stolen," or reserved, for its use. There are two distinct uses for this RAM.

1. Temporary scratch-pad area for the current routine.
2. Permanently-reserved RAM.

For temporary use of RAM, the binary program or ROM can call system routine RESMEM, which will reserve memory. (See the RESMEM system routine in section 7 for documentation.)

RAM can be permanently reserved by a ROM or by a binary program.

RAM RESERVED BY A ROM

RAM that is permanently "stolen" by a ROM must be reserved at power-on. This can be performed during initialization by an INIT routine such as the one shown here:

```
INIT.  BIN
       LDBD 36, = ROMFL          Get ROMFL contents
       JNZ NOTPWO                Jump if not power-on.
       LDMD 36, = FWUSER         Get address of first user byte.
       STMD 36, = UNBAS1         Store base address for later use.
       ADM 36, = 100,0           Add number of bytes needed.
       STMD R36, = FWUSER        Reset the first word pointer.
       JSB = ROMJSB  }           Call the system scratch routine
       DEF SCRAT+    }           to clean up some pointers and the
       BYT Ø         }           program header.
       RTN                       Return.  (Or do more initialization.)
```

System addresses UNBAS1 and UNBAS2 are locations where the base address of reserved RAM is stored. Any time access to this "stolen" RAM is required, the address in UNBAS1 (or UNBAS2) can be loaded into a register and used as a base address with which to index the reserved RAM. For example:

```
       LDMD 22, = UNBAS1     }   Stores zeros into the
       CLM R40               }   10th through the 17th (octal)
       STMD R40, X22, VALUE  }   bytes of stolen RAM.
VALUE  EQU 10
```

RAM RESERVED BY A BINARY PROGRAM

A binary program is not loaded at power-on, so it cannot reserve RAM at this time. Also, a binary program should not reserve memory at the time LOADBIN is performed because a BASIC program may be resident in that RAM space. However, a binary program can reserve RAM within its own program space. For example:

```
VALUE  BSZ 10                    Generates 8 bytes of storage area
                                 and inserts them into object code.
          .
          .
          .
ENTRY. LDMD R22, = BINTAB        Base address of binary program.
       CLM R40
       STMD R40, X22, VALUE      Stores 8 zeros into location VALUE.
```

This routine reserves eight bytes of zeros for permanent use as either scratch-pad or permanent storage memory.

ACCESSING THE PROGRAM CONTROL BLOCK

Although most of the program control block of a BASIC program is of little use to assembly-language programmers, there is one byte that contains program information that can prove valuable in writing binary programs or ROMs. The seventh byte of the PCB contains the status information shown below.

C	O	A	x	x		P	

LEGEND

C = Common Variables
  Ø if no common variables are present
  1 if common variables present in program

O = Option Base
  Ø for option base 1
  1 for option base Ø

A = Allocation Status
  Ø if deallocated program
  1 if allocated program

P = Program Type
  Ø BASIC main program
  1 BASIC subprogram
  2 Assembly-language program (ROM or binary)

Access to this byte can be gained through the section of code shown here:

```
LDMD R30, = FWCURR            Pointer to 1st byte of
                              PCB of current program.
ADM R30, = 6,0
LDBD R30, R30
```

ASSEMBLING

To assemble a binary or ROM program:

1. Ensure that a tape cartridge is inserted in or a disc attached to the HP-83/85.

2. Store the source code on the mass storage device first. This step is not required, but is highly recommended. The HP-83/85 system is vulnerable to object code which takes over hooks or keywords, and source code may be irretrievably lost during assembly. (See below.) Source code is stored with the ASTORE command.

3. Type ASSEMBLE "file name" to assemble the object code on the mass storage device and load it back into memory. Or type ASSEMBLE "file name", number other than Ø to assemble the object code on the mass storage device without loading it into the computer's memory.

The file names used can be different from those specified by NAM. However, a good convention is to name the object code file with the name specified by NAM, followed by a "B" for binary. The source code file can be specified with the name followed by "S" for source.

Generally, the source code will be destroyed during assembly by any of the following conditions:

1. If a LNK has been specified, the linking code will destroy the previous code.

2. If an immediate load is specified and the initialization routine contains faulty code.

3. If a binary or ROM program that takes over CHIDLE is assembled, then listed with the [LIST] key.

USING A BINARY OR ROM PROGRAM

BINARY PROGRAM

Once assembled and loaded, a binary program makes all its keywords available for use by the HP-83/85 system. The keywords become part of the computer's BASIC instruction set, so a BASIC function such as FTOC, for example, could be used as a calculator mode statement:

```
FTOC(32)
```

Or as a BASIC language element:

```
10 LET A = FTOC(100)
```

ROM PROGRAM

A ROM program is stored in a tape or disc file as a series of 125-character ASCII strings. To create an EPROM, the HP-83/85 can be connected through HP-IB (Hewlett-Packard Interface Bus) or another I/O interface card to a commercial PROM burner. The HP-83/85 can then be loaded with a simple BASIC program to read the strings from the tape or disc and send them byte-by-byte to the PROM burner.

NOTE

For further aid in writing binary and ROM programs, study the sample programs supplied on the tape cartridge and disc and listed in section 8 of this manual.

NOTES

SECTION 7

HP-83/85 SYSTEM ROUTINES

This section of the manual gives a listing of the global file contained on the tape cartridge and disc provided with the HP-83/85 Assembler ROM; it also gives detailed information on operation of many specific areas in the computer and on the system routines within the global file.

THE GLOBAL FILE

The global file on the tape cartridge and disc is listed below. It gives the permanent addresses in memory of many of the system routines used by the HP-83/85. The global file also contains locations of system pointers, buffers, variables, and constants which may be referenced in a binary program.

On the tape cartridge and disc supplied with the Assembler ROM, there are actually two copies of this global file.

--GL01S and GL02S together make up the global source file. This is an extended file, type ****, and can be edited by the user, if a user-written change to the global file is desired. GL01S and GL02S can also be used to print out a listing of the global file.

--GLOBAL is the global file in object code. This is a data file containing normal ASCII strings that make up the assembled object code for the global file. When the pseudo-op GLO GLOBAL has been placed near the beginning of a binary program, during assembly the computer will look at this file for the addresses of any undefined labels in the program.

Although it is usually more convenient, it is not necessary to use the file GLOBAL as a label table. You may create your own label table on a mass storage device, or you may specify the addresses of the system routines called in a binary program by adding them to the label table within the program.

The global file on the following pages is the same as the one on the tape cartridge and disc supplied with the Assembler ROM.

LEGEND

Name Name of routine, buffer, etc.

Address Permanent octal address of routine in HP-83 or HP-85 memory.

Description A short description of the routine.

GLOBAL FILE

NAME ADDRESS DESCRIPTION

```
10 !***************************
15 !*                         *
20 !*    HP-83/85 ASSEMBLER   *
30 !*       GLOBAL FILE       *
40 !* (c) Hewlett-Packard Co. *
50 !*          1980           *
55 !*                         *
60 !***************************
70        GLO
80 FWUSER DAD 100000          !FWA USER AREA
90 FWPRGM DAD 100002          !FWA PROGRAM AREA
100 FWCURR DAD 100004         !PTR TO CURRENT PGM
110 NXTMEM DAD 100006         !NEXT IN AVAIL USER MEM
120 LAVAIL DAD 100010         !LAST AVAIL BYTE IN PGM AREA
130 CALVRB DAD 100012         !START OF CALC VARIABLES
140 RTNSTK DAD 100014         !TOP OF GOSUB RETURN STACK
150 NXTRTN DAD 100016         !NEXT AVAIL GOSUB/RTN
160 FWBIN  DAD 100020         !=LWAMEM IF NO BPGM LOADED ELSE =BINTAB-1
170 LWAMEM DAD 100022         !LWA USER MEM
180 LLDCOM DAD 100025         !LAST LINE DECOMPILE
190 FLDCOM DAD 100027         !FIRST LINE DECOMPILE
200 DISPTR DAD 100033         !DISP BUFFER PTR
210 PRTPTR DAD 100035         !PRINT BUFFER PTR
220 ONFLAG DAD 100040         !ON GOSUB FLAG
230 AUTO#  DAD 100054         !AUTO LINE # LAST VAL
240 AUTOI  DAD 100056         !AUTO LINE # INCREMENT
250 ERLIN# DAD 100062         !LINE# OF BAD LINE
260 ERNUM# DAD 100064         !ERROR NUMBER
270 ERRROM DAD 100065         !ROM# OF ERROR
280 ERROM# DAD 100066         !ROM # OF LAST ERROR
290 EDMOD2 DAD 100067         !INS/RPL MODE FLAG
300 ERRORS DAD 100070         !RUN TIME ERRORS
310 ERRTYP DAD 100071         !ERROR TYPE
320 KEYCNT DAD 100120         !KEYBOARD COUNTER RPT
330 KRPET1 DAD 100121         !MAJOR KYBD REPEAT
340 KRPET2 DAD 100122         !MINOR KYBD REPEAT
350 LDFLTR DAD 100123         !LIST BREAK LINE COUNT
360 DRG    DAD 100125         !DEG/RAD/GRAD
370 SVCWRD DAD 100151         !SERVICE WORD
380 IOSW   DAD 100152         !IO SVC WORD
390 CRTBYT DAD 100176         !CRT BYTE ADDRESS
400 CRTRAM DAD 100200         !CRT PAGE ADDRESS
410 XMAP   DAD 100262         !LAST X PLOTTED (0-255)
420 YMAP   DAD 100263         !LAST Y PLOTTED (0-255)
430 CS.C.  DAD 100264         !CRT IS select code (8 BYTES)
440 PS.C.  DAD 100274         !PRINTER IS select code
```

NAME ADDRESS DESCRIPTION

```
450 INPBUF DAD 100310          !PARSER INPUT BUFFER
460 LASTIN DAD 100447          !END OF INPUT BUFFER
470 ERRBUF DAD 100450          !ERROR BUFFER (44 BYTES)
480 ERBEND DAD 100524          !END BUFFER +1
490 PRTBUF DAD 100524          !PRINT BUFFER
500 DISBUF DAD 100564          !DISPLAY BUFFER
510 PCR    DAD 100642          !BASIC PGM LINE PTR
520 PRFLAG DAD 100644          !PRINTED YET? FLAG AT PRINT EOL FOR PRINT
530 DSFLAG DAD 100645          !PRINTED YET? FLAG AT PRINT EOL FOR DISP
540 TIME   DAD 100650          !TIME OF DAY
550 DATE   DAD 100660          !JULIAN DAY YEAR
560 DISPLN DAD 100665          !1 BYTE DISPLAY LINE LEN
570 PRNTLN DAD 100666          !1 BYTE PRINTER LINE LEN
580 KEYHIT DAD 100671          !KEYBOARD ASCII CODE
590 INPTR  DAD 100672          !INPUT LINE POINTER
600 LEGEND DAD 100710          !KEY LABEL LEGEND AREA
610 LEGEN2 DAD 100750          !SECOND LINE LEGEND AREA
620 CRTWRS DAD 101016          !CRT STATUS IN RAM
630 P.BUFF DAD 101075          !INDIRECT BUFFER PTR
640 P.PTR  DAD 101077          !INDIRECT PTR TO BYTE COUNT FOR CURRENT BUFFER
650 P.FLAG DAD 101101          !INDIR. PTR TO PRFLAG OR DSFLAG OR P/P ROM FLAG
660 LINELN DAD 101103          !DEVICE LINE LENGTH
670 SCTEMP DAD 101110          !SELECT CODE TEMP STORE
680 STSIZE DAD 101130          !STATEMENT SIZE PLACE HOLDER PTR
690 TOS    DAD 101132          !TOP OF R12 STACK
700 ROMFL  DAD 101231          !ROM FLAG FOR INIT ROUTINES
710 BINTAB DAD 101233          !CONTAINS BASE ADDRESS OF BPGM
720 ROMTAB DAD 101235          !LIST OF PRESENT EXTERNAL ROMS
730 ROMLST DAD 101272          !LAST ENTRY IN ROM TABLE
740 STACK  DAD 101300          !BEGINNING OF THE R6 STACK
750 !*************************
760 !* THE R6 STACK USES THE *
770 !* AREA OF MEMORY FROM   *
780 !* 101300 THRU 101777.   *
790 !*************************
800 IOTRFC DAD 102400          !I/O TRAFFIC INTERCEPT
810 IOSP   DAD 102407          !I-O INTERRUPT SERVICE PTR
820 CHIDLE DAD 102416          !CHAR. EDITOR INTERCEPT
830 KYIDLE DAD 102425          !KEYBOARD INTERRUPT INTERCEPT
840 RMIDLE DAD 102434          !EXEC LOOP INTERCEPT
850 IMERR  DAD 102452          !IMAGE ERROR INTERCEPT
860 PRSIDL DAD 102461          !PARSER INTERCEPT
870 IRQ20  DAD 102470          !I-O INTERRUPT
880 SPARO  DAD 102512          !SYSTEM MONITOR INTERRUPT HOOK
890 SPAR1  DAD 102523          !SPARE INTERRUPT HOOK #1
900 !**************************
910 !* THE FOLLOWING LOCATIONS*
920 !* CONTAIN BASE ADDRESSES *
930 !* OF STOLEN RAM FOR EACH *
940 !* OF THE EXTERNAL ROMS.  *
950 !**************************
960 IOBASE DAD 102536          !I/O ROM
970 MSBASE DAD 102540          !MASS STORAGE ROM
980 AGLBAS DAD 102542          !PLOTTER/PRINTER ROM
990 APRBAS DAD 102544          !ADVANCED PROGRAMMING ROM
1000 BSRBAS DAD 102546         !BLUE SPRUCE
1010 MBASE  DAD 102550         !MATRIX ROM
1020 ASMBAS DAD 102552         !ASSEMBLER ROM
```

NAME	ADDRESS	DESCRIPTION

```
1030 UNBAS1 DAD 102554          !UNUSED: AVAILABLE
1040 UNBAS2 DAD 102556          !UNUSED: AVAILABLE
1050 FWROM  EQU 103300          !FWA USER PROGRAM ROMRAM
1060 !*************************
1065 !*                       *
1070 !*  I/O ADDRESSES        *
1075 !*                       *
1080 !*************************
1090 GINTEN DAD 177400          !GLOBAL INTERRUPT ENABLE
1100 GINTDS DAD 177401          !GLOBAL INTERRUPT DISABLE
1110 KEYSTS DAD 177402          !KEYBOARD STATUS
1120 KEYCOD DAD 177403          !KEYBOARD CODE AND EOJOB
1130 CRTSAD DAD 177404          !CRT START ADDRESS
1140 CRTBAD DAD 177405          !CRT BYTE ADDRESS
1150 CRTSTS DAD 177406          !CRT STATUS
1160 CRTDAT DAD 177407          !CRT DATA
1170 RSELEC DAD 177430          !ROM SELECT ADDRESS
1180 !*************************
1190 !* THE FOLLOWING ARE ONLY*
1200 !* CONVENIENT LABELS FOR *
1210 !* SOME ASCII CODES AND  *
1220 !* SOME DIGITS           *
1230 !*************************
1240 ZRO    EQU 0
1250 ONE    EQU 1
1260 TWO    EQU 2
1270 THREE  EQU 3
1280 FOUR   EQU 4
1290 FIVE   EQU 5
1300 SIX    EQU 6
1310 SEVEN  EQU 7
1320 EIGHT  EQU 10
1330 NINE   EQU 11
1340 TEN    EQU 12
1350 BLANK  EQU 40
1360 BANG   EQU 41
1370 "      EQU 42
1380 #      EQU 43
1390 $      EQU 44
1400 %      EQU 45
1410 &      EQU 46
1420 '      EQU 47
1430 (      EQU 50
1440 )      EQU 51
1450 *      EQU 52
1460 +      EQU 53
1470 ,      EQU 54
1480 -      EQU 55
1490 .      EQU 56
1500 /      EQU 57
```

```
:        EQU 72
;        EQU 73
<        EQU 74
=        EQU 75
>        EQU 76
?        EQU 77
@        EQU 100
A        EQU 101
B        EQU 102
C        EQU 103
D        EQU 104
E        EQU 105
F        EQU 106
G        EQU 107
H        EQU 110
I        EQU 111
J        EQU 112
K        EQU 113
L        EQU 114
M        EQU 115
N        EQU 116
O        EQU 117
P        EQU 120
Q        EQU 121
R        EQU 122
S        EQU 123
T        EQU 124
U        EQU 125
V        EQU 126
W        EQU 127
X        EQU 130
Y        EQU 131
Z        EQU 132
[        EQU 133
\        EQU 134
]        EQU 135
^        EQU 136
_        EQU 137
`        EQU 140
a        EQU 141
b        EQU 142
c        EQU 143
d        EQU 144
e        EQU 145
f        EQU 146
g        EQU 147
h        EQU 150
i        EQU 151
j        EQU 152
k        EQU 153
```

```
2010 l       EQU 154
2020 m       EQU 155
2030 n       EQU 156
2040 o       EQU 157
2050 p       EQU 160
2060 q       EQU 161
2070 r       EQU 162
2080 s       EQU 163
2090 t       EQU 164
2100 u       EQU 165
2110 v       EQU 166
2120 w       EQU 167
2130 x       EQU 170
2140 y       EQU 171
2150 z       EQU 172
2160         LNK GLO2S
```

NAME ADDRESS DESCRIPTION

```
10 !**************************
20 !*   HP-83/85 ASSEMBLER    *
30 !*       GLOBAL FILE       *
40 !*       SECTION   2       *
50 !* (c) Hewlett-Packard Co. *
60 !*          1980           *
70 !**************************
80 !
90 !
2160 !***********************
2170 !* SYSTEM ROUTINE ENTRY *
2180 !*   POINT ADDRESSES    *
2190 !***********************
2200 ABS5   DAD 53731        !ABSOLUTE VALUE
2210 ADDROI DAD 52150        !ADD TWO NUMBERS
2220 ALFA   DAD 11775        !CHECK FOR ALPHA CHAR. & UPC IF SO
2230 ALPHA. DAD 36105        !FORCE CRT TO ALPHA MODE
2240 ASIGN. DAD 27056        !OPEN A BUFFER TO A DATA FILE
2250 ATN2.  DAD 76455        !DOES ATN2(Y,X)
2260 BEEP.  DAD 6737         !BEEP COMMAND
2270 BLKLIN DAD 36320        !BLANK A LINE ON CRT
2280 BPLOT. DAD 34365        !BPLOT
2290 BYTCRT DAD 35423        !SETS CRT BYTE ADDRESS TO R#
2300 BYTCR! DAD 35422        !SETS CRT BYTE ADDRESS TO R34
2310 CEIL10 DAD 53615        !CEIL FUNCTION
2320 CHKSTS DAD 36335        !DEMAND CRT NOT BUSY
2330 CHSROI DAD 52075        !CHANGE SIGN OF A REAL OR INTEGER
2340 CLEAR. DAD 35021        !CLEAR A PAGE OF CRT ALPHA
2350 CLREOL DAD 35535        !CLEAR TO END OF LINE
2360 CNTRTR DAD 36002        !COUNT CRT RETRACES
2370 COMFLT DAD 32621        !COMPARE TWO NUMBERS
2380 COMMA$ DAD 70634        !PRINT A STRING FOLLOWED BY COMMA
2390 COMMA. DAD 70756        !PRINT A NUMBER FOLLOWED BY A COMMA
2400 CONBIN DAD 3572         !CONVERT A 16-BIT # TO A REAL #
2410 CONCA. DAD 75005        !CONCATENATE TWO STRINGS
2420 CONINT DAD 44321        !CONVERT A REAL # TO A 16-BIT
2430 COPY.  DAD 75360        !COPY CRT TO INTERNAL PRINTER
2440 COS10  DAD 53556        !COSINE FUNCTION
2450 COT10  DAD 53536        !COTANGENT FUNCTION
2460 CREAT. DAD 26561        !CREATE A DATA FILE
2470 CRTBL+ DAD 36255        !INITIALIZE PART OF CRT ALPHA
2480 CRTBLK DAD 36247        !INITIALIZE ALL OF CRT ALPHA
2490 CRTINT DAD 36177        !INITIALIZE ALL OF ALPHA & GRAPHICS
2500 CRTPOF DAD 35703        !POWER DOWN CRT
2510 CRTPUP DAD 35716        !POWER UP CRT
2520 CRTUNW DAD 36067        !UNWIPE CRT
2530 CRTWPO DAD 35661        !WIPE-OUT CRT TO HIDE UGLY FLASH
2540 CSEC10 DAD 53503        !COSECANT FUNCTION
2550 CURS   DAD 35055        !SPIT OUT A CURSOR TO CRT
2560 CVNUM  DAD 71135        !FORMAT A REAL # TO ASCII FOR OUTPUT
2570 DATE.  DAD 37673        !DATE FUNCTION
2580 DECUR2 DAD 35547        !ERASE CURSOR FROM CRT
2590 DEG.   DAD 61736        !SET HP-85 TO DEGREE MODE
2600 DEG10  DAD 54142        !RADIANS TO DEGREE CONVERSION
2610 DIGIT  DAD 12027        !CHECK FOR A DIGIT
2620 DISP.  DAD 70046        !SET PRINT PTRS TO 'CRT IS' DEVICE
2630 DIV2   DAD 51641        !DIVIDE TWO NUMBERS
2640 DMNDCR DAD 15060        !DEMAND EITHER A CARRIAGE RTN OR BANG (!)
2650 DNCUR. DAD 35306        !MOVE CURSOR DOWN ONE ROW ON CURRENT PAGE
2660 DNCURS DAD 35370        !MOVE CURSOR DOWN ON ALL 4 PAGES
```

NAME ADDRESS DESCRIPTION

```
2670 DRAW.  DAD 33015          !DRAW A LINE ON THE CRT
2680 DRV12. DAD 5462           !DUMP A BUFFER TO CRT,PRINTER,OR I/O
2690 EOJ2   DAD 34772          !RESET R17 AND SVCWRD AFTER KEY IS HANDLED
2700 EPS10  DAD 54126          !EPSILON FUNCTION
2710 EQ.    DAD 62173          !CHECK TWO #'S FOR EQUALITY
2720 EQ$.   DAD 3006           !CHECKS TWO STRINGS FOR EQUALITY
2730 ERROR  DAD 6615           !REPORTS AN ERROR
2740 ERROR+ DAD 6611           !REPORTS ERROR & THROWS AWAY ONE RETURN
2750 EXP5   DAD 52377          !EXPONENTIATE
2760 FETAV  DAD 44727          !FETCH ARRAY VARIABLE
2770 FETAVA DAD 44734          !FETCH ARRAY VARIABLE ADDRESS
2780 FETST  DAD 45206          !FETCH STRING VARIABLE
2790 FETSV  DAD 44535          !FETCH SIMPLE NUMERIC VARIABLE
2800 FETSVA DAD 44556          !FETCH SIMPLE VARIABLE ADDRESS
2810 FLIP.  DAD 35011          !FLIP KEYBOARD UPPERCASE/LOWERCASE
2820 FORMN+ DAD 71146          !FORMAT NUMBER
2830 FP5    DAD 54071          !FRACTIONAL PART
2840 G$N    DAD 14323          !GET A STRING AND NUMBER
2850 G$N+NN DAD 14421          !GET A STRING AND NUMBER AND OPTIONS
2860 G012N  DAD 14465          !GET 0,1,OR 2 NUMBERS
2870 G01N   DAD 14504          !GET 0 OR 1 NUMBERS
2880 G0OR2N DAD 14522          !GET 0 OR 2 NUMBERS
2890 G12OR4 DAD 14550          !GET 1,2 OR 4 NUMBERS
2900 G1OR2N DAD 14537          !GET 1 OR 2 NUMBERS
2910 GCHAR  DAD 11755          !GET THE NEXT CHAR TO R20
2920 GCLR.  DAD 36013          !GCLEAR
2930 GEQ.   DAD 62304          !CHECK TWO #'S FOR >=
2940 GEQ$.  DAD 3111           !CHECK STRINGS FOR >=
2950 GET$N? DAD 14560          !GET STRING AND NUMBER?
2960 GET)   DAD 13365          !GET CLOSE PAREN
2970 GET1$  DAD 14455          !GET ONE STRING
2980 GET1N  DAD 14337          !GET 1 NUMBER
2990 GET2N  DAD 14407          !GET 2 NUMBERS
3000 GET4N  DAD 14414          !GET 4 NUMBERS
3010 GETCMA DAD 13414          !DEMAND A COMMA
3020 GETCM? DAD 13425          !CHECK FOR A COMMA
3030 GETPA? DAD 14516          !GET PARAMETERS
3040 GETPAR DAD 14342          !GET PARAMETERS
3050 GRAD.  DAD 61753          !SET COMPUTER TO GRAD TRIG MODE
3060 GR.    DAD 62255          !CHECK TWO NUMBERS FOR >
3070 GR$.   DAD 3036           !CHECK TWO STRINGS FOR >
3080 GRAPH. DAD 36147          !FORCE CRT TO GRAPH MODE
3090 GRINIT DAD 36220          !INITIALIZE THE GRAPHICS SCREEN
3100 HLFLIN DAD 35121          !DUMP A BUFFER TO THE CRT WITH NO CR
3110 HMCURS DAD 35527          !SEND CURSOR TO HOME
3120 ICOS   DAD 76552          !ARCCOSINE FUNCTION
3130 IDRAW. DAD 32752          !INCREMENTAL DRAW
3140 IMOVE. DAD 31675          !INCREMENTAL MOVE
3150 INCHR  DAD 35244          !READ IN A CHARACTER FROM CRT
3160 INCHR- DAD 35220          !READ CRT IF WPO GUARANTEED
3170 INF10  DAD 53524          !INFINITY
3180 INT5   DAD 53776          !INTEGER PART
3190 INTDIV DAD 54005          !INTEGER DIVIDE
3200 INTEGR DAD 11447          !GET AN INTEGER NUMBER
3210 INTMUL DAD 53076          !MULTIPLY TWO 16-BIT BINARY NUMBERS
3220 INTORL DAD 56343          !CONVERT A TAGGED INTEGER TO A REAL #
3230 IP5    DAD 54174          !INTEGER PART
3235 ISIN   DAD 76542          !ARCSIN FUNCTION
3240 ITAN   DAD 76562          !ARCTANGENT
3250 LABEL. DAD 34044          !LABEL ON CRT GRAPHICS
3260 LDIR.  DAD 34020          !SET LABEL DIRECTION
```

	NAME	ADDRESS	DESCRIPTION
3270	LEQ.	DAD 62232	!CHECK TWO #'S FOR <=
3280	LEQ$.	DAD 3100	!CHECK TWO STRINGS FOR <=
3290	LN5	DAD 51551	!NATURAL LOGARITHM
3300	LOGT5	DAD 51720	!LOG BASE TEN
3310	LT.	DAD 62213	!CHECK TWO #'S FOR <
3320	LT$.	DAD 3057	!CHECK TWO STRINGS FOR <
3330	LTCUR.	DAD 35332	!MOVE CURSOR LEFT ONE COLUMN ON CURRENT PAGE
3340	LTCURS	DAD 35376	!MOVE CURSOR LEFT ON ALL 4 PAGES
3350	MAX10	DAD 55364	!MAXIMUM FUNCTION
3360	MIN10	DAD 55345	!MINIMUM FUNCTION
3370	MOD10	DAD 51744	!MOD FUNCTION
3380	MOVCRS	DAD 35410	!MOVE CURSOR
3390	MOVDN	DAD 37324	!MOVE MEMORY AND DECREMENT
3400	MOVE.	DAD 31703	!MOVE ON CRT
3410	MOVUP	DAD 37365	!MOVE MEMORY AND INCREMENT
3420	MPYROI	DAD 52722	!MULTIPLY TWO NUMBERS
3430	NARRE+	DAD 13376	!SCAN & PARSE ARRAY REF WITHOUT PARENS
3440	NARREF	DAD 13402	!PARSE ARRAY REF WITHOUT PARENS
3450	NUMCON	DAD 13466	!GET A NUMERIC CONSTANT
3460	NUMVA+	DAD 12407	!SCAN AND GET A NUMERIC VALUE
3470	NUMVAL	DAD 12412	!GET A NUMERIC VALUE
3480	OFTIM.	DAD 66211	!TURN A TIMER OFF
3490	ONEB	DAD 56113	!GET 1 NUMBER OFF R12 AS 15-BIT SIGNED BINARY
3500	ONEI	DAD 56154	!GET ONE NUMBER OFF R12 AS TAGGED INTEGER
3510	ONER	DAD 56215	!GET 1 NUMBER OFF R12 AS FLOATING POINT
3520	ONEROI	DAD 56253	!GET 1 NUMBER OFF R12 AS REAL OR INTEGER
3530	ONTIM.	DAD 66041	!TURN ON A TIMER
3540	OUTCHR	DAD 35114	!OUTPUT ONE CHAR TO CRT
3550	OUTSTR	DAD 35052	!OUTPUT A STRING TO CRT
3560	P#ARAY	DAD 57642	!PRINT AN ARRAY TO A DATA FILE
3570	PAPER.	DAD 76144	!ADVANCE INTERNAL PRINTER
3580	PEN.	DAD 66416	!PEN STATEMENT
3590	PENUP.	DAD 66440	!PENUP
3600	PI10	DAD 53577	!PI FUNCTION
3610	PLOT.	DAD 32642	!PLOT TO CRT
3620	POS.	DAD 3435	!POS FUNCTION
3630	PRDVR1	DAD 75767	!OUTPUT A STRING TO THE INTERNAL PRINTER
3640	PRINT.	DAD 70067	!SET UP PRINT PTRS TO 'PRINTER IS' DEVICE
3650	PRLINE	DAD 70402	!DUMP THE PRINT BUFFER
3660	PRNT#$	DAD 30577	!PRINT A STRING TO A DATA FILE
3670	PRNT#.	DAD 30055	!MOVE THE PRINT PTRS IN THE BUFFER
3680	PRNT#N	DAD 31022	!PRINT A NUMBER TO A DATA FILE
3690	PURGE.	DAD 26013	!PURGE FILES
3700	PUSH1A	DAD 14244	!PUSH A TOKEN
3710	PUSH32	DAD 14277	!PUSH TOKEN IN R14 & REGS R44-6 & SCAN
3720	PUSH45	DAD 14266	!PUSH TOKEN IN R14 & REGS R44-5 & SCAN
3730	R#ARAY	DAD 77602	!READ AN ARRAY FROM A DATA FILE
3740	RAD.	DAD 61746	!PUT COMPUTER IN RADIANS TRIG MODE
3750	RAD10	DAD 53675	!DEGREES TO RADIANS CONVERSION
3760	READ#$	DAD 31335	!READ A STRING FROM A DATA FILE
3770	READ#.	DAD 30055	!MOVE THE READ PTR
3780	READ#N	DAD 31167	!READ A NUMBER FROM A DATA FILE
3790	REFNUM	DAD 17025	!GET A VARIABLE REFERENCE
3800	RELMEM	DAD 37534	!RELEASE RESERVED MEMORY
3810	REM10	DAD 51736	!REMAINDER
3820	RESMEM	DAD 37442	!RESERVE MEMORY FOR TEMPORARY SCRATCH
3830	RND10	DAD 53144	!RANDOM NUMBER FUNCTION
3840	RNDIZ.	DAD 55115	!RANDOMIZE STATEMENT
3850	ROMJSB	DAD 4776	!FOR CALLING BETWEEN BANK SELECTED ROMS
3860	ROMRTN	DAD 4762	!GTO ROMRTN = RETURN WITH ROM 0 SELECTED
3870	ROU10	DAD 55163	!ROUND

NAME ADDRESS DESCRIPTION

```
3875 RSMEM- DAD 37453          !RESERVE TEMPORARY SCRATCHPAD MEMORY
3880 RSUM#K DAD 37726          !CHECKSUM # OF BYTES
3890 RSUM8K DAD 37722          !CHECKSUM AN 8K ROM
3900 RTCUR. DAD 35351          !MOVE CURSOR RIGHT ON CURRENT PAGE
3910 RTCURS DAD 35404          !MOVE CURSOR RIGHT ON ALL 4 PAGES
3920 RTOIN  DAD 44204          !CONVERT A REAL # TO A TAGGED INTEGER
3930 SCALE. DAD 66247          !SCALE THE CRT GRAPHICS
3940 SCAN   DAD 11262          !SCAN FOR PARSER
3950 SCAN+  DAD 11257          !GCHAR AND SCAN
3960 SCRAT+ DAD 4344           !SUBSET OF SCRAT. (SCRATCHES BASIC PGM & BPGM)
3970 SCRAT. DAD 4437           !SCRATCH (DOES SCRAT+ & RESETS SOME PTRS)
3980 SCRDN  DAD 35625          !SCROLL ALPHA DOWN
3990 SCRUP  DAD 35654          !SCROLL ALPHA CRT UP
4000 SEC10  DAD 53463          !SECANT
4010 SEMIC$ DAD 70643          !PRINT A STRING FOLLOWED BY SEMICOLON
4020 SEMIC. DAD 70765          !PRINT A NUMBER FOLLOWED BY A SEMICOLON
4030 SEQNO+ DAD 17454          !PUSH THE INCOMING TOKEN AND GET A LINE #
4040 SEQNO  DAD 17457          !GET A LINE NUMBER
4050 SET240 DAD 11243          !SET IMMEDIATE BREAK BITS IN R17
4060 SGN5   DAD 53405          !SIGN FUNCTION
4070 SIN10  DAD 53546          !SINE
4080 SMLINT DAD 13474          !PARSE AN INTEGER
4090 SQR5   DAD 52442          !SQARE ROOT
4100 STBEEP DAD 7017           !STANDARD BEEP (NO PARAMETERS)
4110 STOST  DAD 45603          !STORE STRING
4120 STOSV  DAD 45254          !STORE SIMPLE AND ARRAY VARIABLE
4130 STRCON DAD 14036          !PARSE A QUOTED STRING
4140 STREX+ DAD 13623          !SCAN AND PARSE A STRING EXPRESSION
4150 STREXP DAD 13626          !PARSE A STRING EXPRESSION
4160 STRREF DAD 13753          !PARSE A STRING VARIABLE AS A STORE STRING
4170 SUBROI DAD 52127          !SUBTRACT TWO NUMBERS
4180 TAN10  DAD 53566          !TANGENT
4190 TIME.  DAD 65517          !TIME FUNCTION
4200 TRY1N  DAD 14566          !GETS 0 OR 1 NUMERIC VALUES
4210 TWOB   DAD 56176          !GET TWO NUMBERS OFF R12 AS 15-BIT SIGNED #'S
4220 TWOR   DAD 56236          !GET TWO NUMBERS OFF R12 AS REAL #'S
4230 TWOROI DAD 56266          !GET TWO NUMBERS OFF R12 AS REAL OR INTEGER
4240 UNEQ$. DAD 3025           !CHECK TWO STRINGS FOR NOT EQUAL
4250 UNEQ.  DAD 62202          !COMPARE TWO #'S FOR INEQUALITY
4260 UNQUOT DAD 14212          !PARSE AN UNQUOTED STRING
4270 UPC$.  DAD 3373           !UPPER CASE FUNCTION
4280 UPCUR. DAD 35264          !MOVE CURSOR UP ON CURRENT PAGE
4290 UPCURS DAD 35362          !MOVE CURSOR UP ON ALL FOR PAGES
4300 VAL$.  DAD 3207           !VAL$ FUNCTION
4310 VAL.   DAD 3250           !VAL FUNCTION
4320 WAIT.  DAD 65701          !WAIT X MILLISECONDS
4330 XAXIS. DAD 32303          !XAXIS STATEMENT
4340 YAXIS. DAD 32347          !YAXIS STATEMENT
4350 YTX5   DAD 53242          !Y^X FUNCTION
4360 ZROMEM DAD 44066          !ZERO OR BLANK A BLOCK OF MEMORY
4370        FIN
```

SYSTEM OPERATION AND ROUTINES

This section provides some specific details, register conventions, etc. for certain areas of HP-83/85 system operation. It also shows the input conditions required and the outputs produced by selected system routines. The names and addresses of most (but not all) of the system routines detailed here are also available on the Global File tape cartridge and disc.

The areas of focus are:
--Parsing and parse routines
--Runtime and runtime routines
--General-purpose utility routines
--CRT control and routines
--Tape control routines
--Decompiling

The system routines are arranged within their areas of primary use. Simply because a routine is listed under a certain application, however, does not limit its use to that area. For example, many utility routines may also be used during runtime operations.

SYSTEM ROUTINE FORMAT

The format of the individual system routines is shown here:

Name: Name of the routine (from the global file).

Address: Permanent octal address of routine in computer memory.

Type: Primary tasks for which routine will be used.

Function: Outlines the function of the routine.

Input Conditions: Shows the assumptions made by the routine (e.g., contents of specific registers and condition of stack pointed to by CPU register R12) when routine is called.

Output Conditions: Shows results, outputs, etc., as they are found in specific registers and/or on the stack addressed by CPU register R12.

NOTE

In the descriptions of R12 stack contents, the contents of the stack are shown as they occur on the stack. The nomenclature R12→ indicates the location of the stack pointer.

CPU Changes: Darkened area indicates the CPU registers whose contents are altered by execution of the routine.

DCM: Setting of decimal mode flag after routine is executed: B indicates binary mode; D indicates decimal mode; — (dash) indicates unchanged by routine; and U indicates undefined.

E: Contents of four-bit extend register after routine is executed. Contents may be: Value (2-digit octal quantity); — (dash) for unchanged by routine; or U for undefined.

DRP: Shows setting of data register pointer after routine is executed. May be: CPU register number; — (dash) for unchanged by routine; or U for undefined.

ARP: Shows setting of address register pointer after routine is executed. May be: CPU register number; — (dash) for unchanged by routine; or U for undefined.

STATUS: Shows whether other CPU status flags are altered. May be: — (dash) for unchanged; or U for undefined.

ROMJSB: Indicates whether or not this routine, if called from an external ROM, must be called through ROMJSB. May be: Y for yes, must be called through ROMJSB; or N for no, need not be called through ROMJSB.

PARSING AND PARSE ROUTINES

PARSE ROUTINE REGISTERS

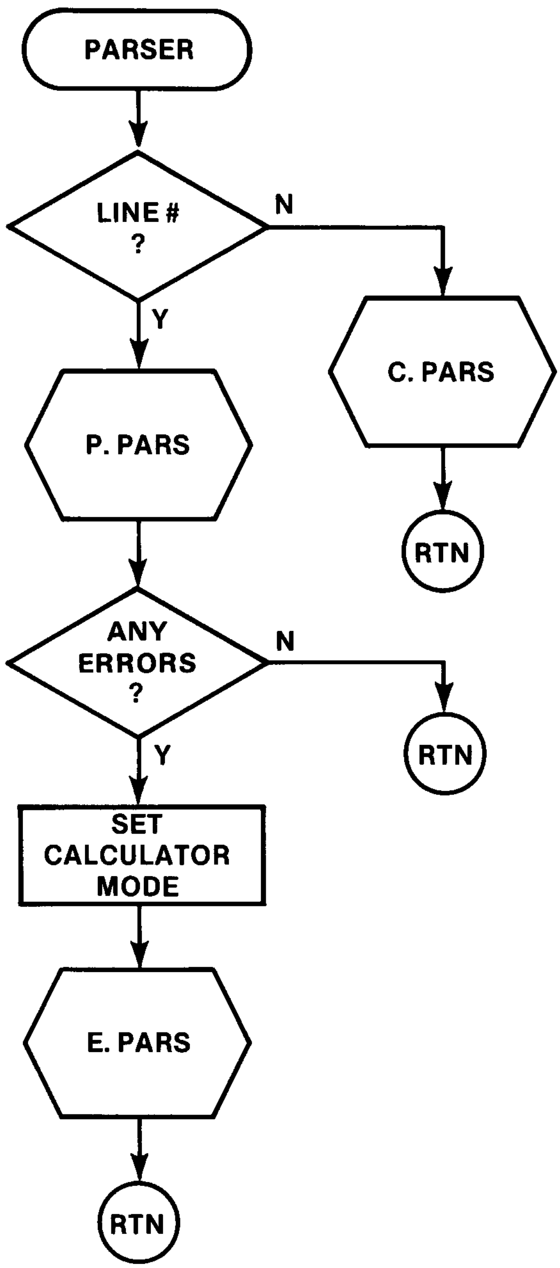
In parsing, the HP-83/85 system uses the CPU registers shown here.

R10		Input buffer pointer.
R12		Output stack pointer.
R14		Next token. (Set by SCAN routine.)
R20		Next non-blank character. (Set by GCHAR routine.)
R40-R47		Detailed scan output. (Set by SCAN.)
R40		First character scanned.
R41-R42		ROM #. (If R42 = Ø.)
	or	Binary program address. (If R42 ≠ Ø.)
	or	System ROM. (If R41 = R42 = Ø.)
R43		ROM token #.
	or	Binary program token #.
	or	Type. (If variable.)
R44-R46		Name. (If variable, R46 not used.)
	or	Integer.
	or	Secondary attributes for function.
R47		Primary attributes.

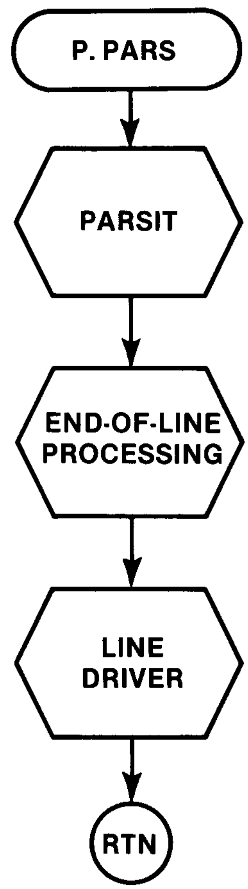
PARSE ROUTINE REGISTER USAGE

PARSING FLOW

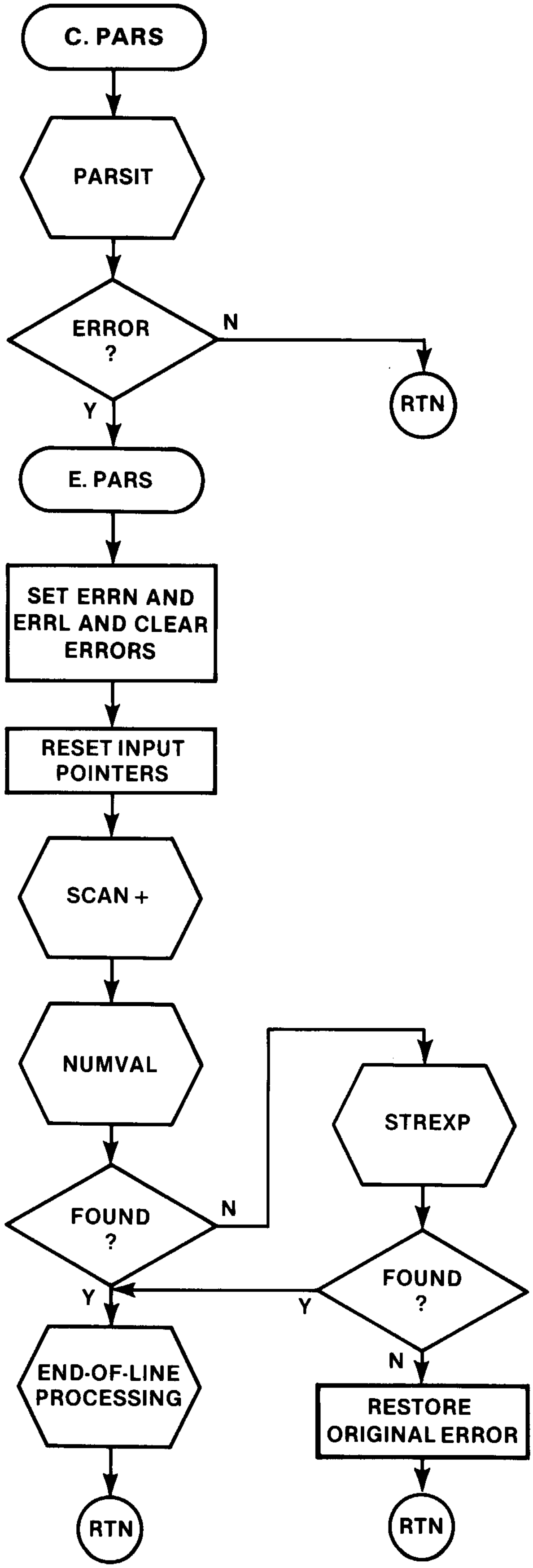
Program flow in parsing is shown in the flowcharts on the next few pages. A brief explanation follows the flowcharts.



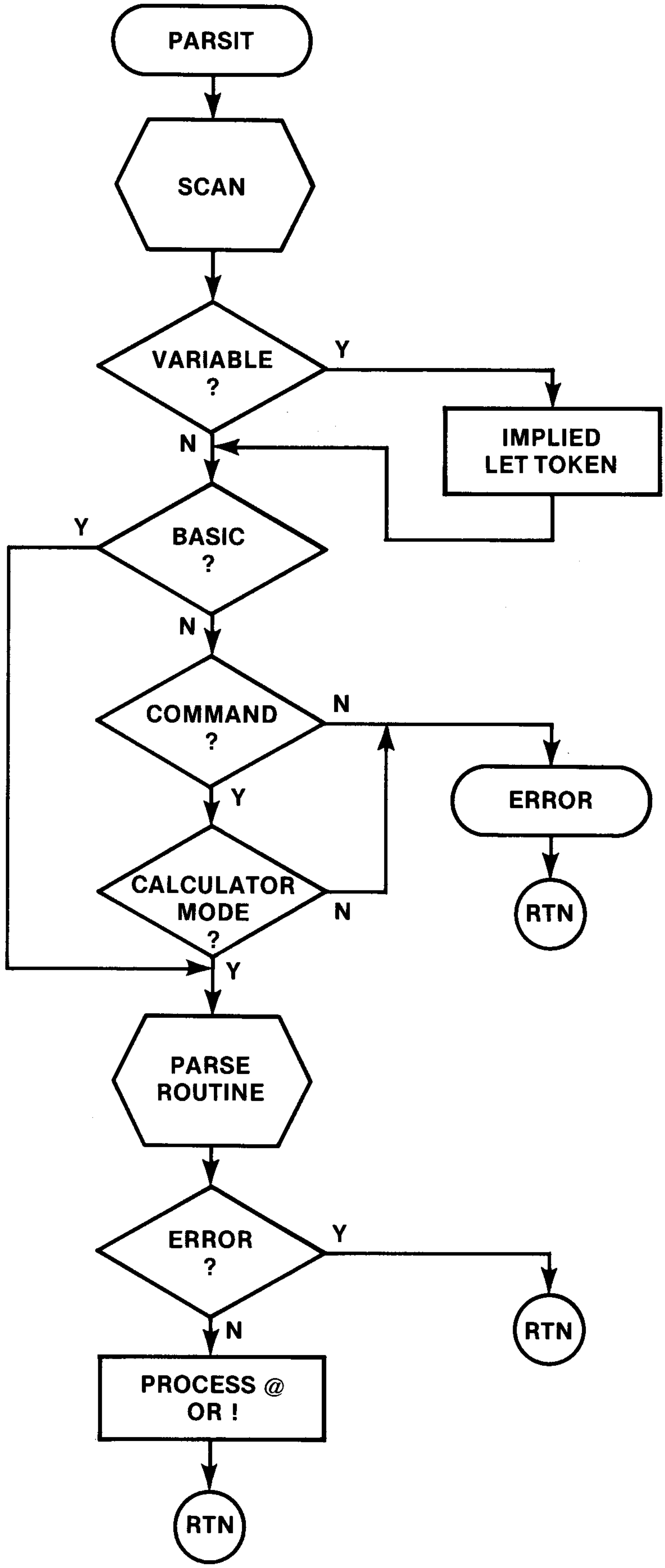
MAIN PARSE LOOP



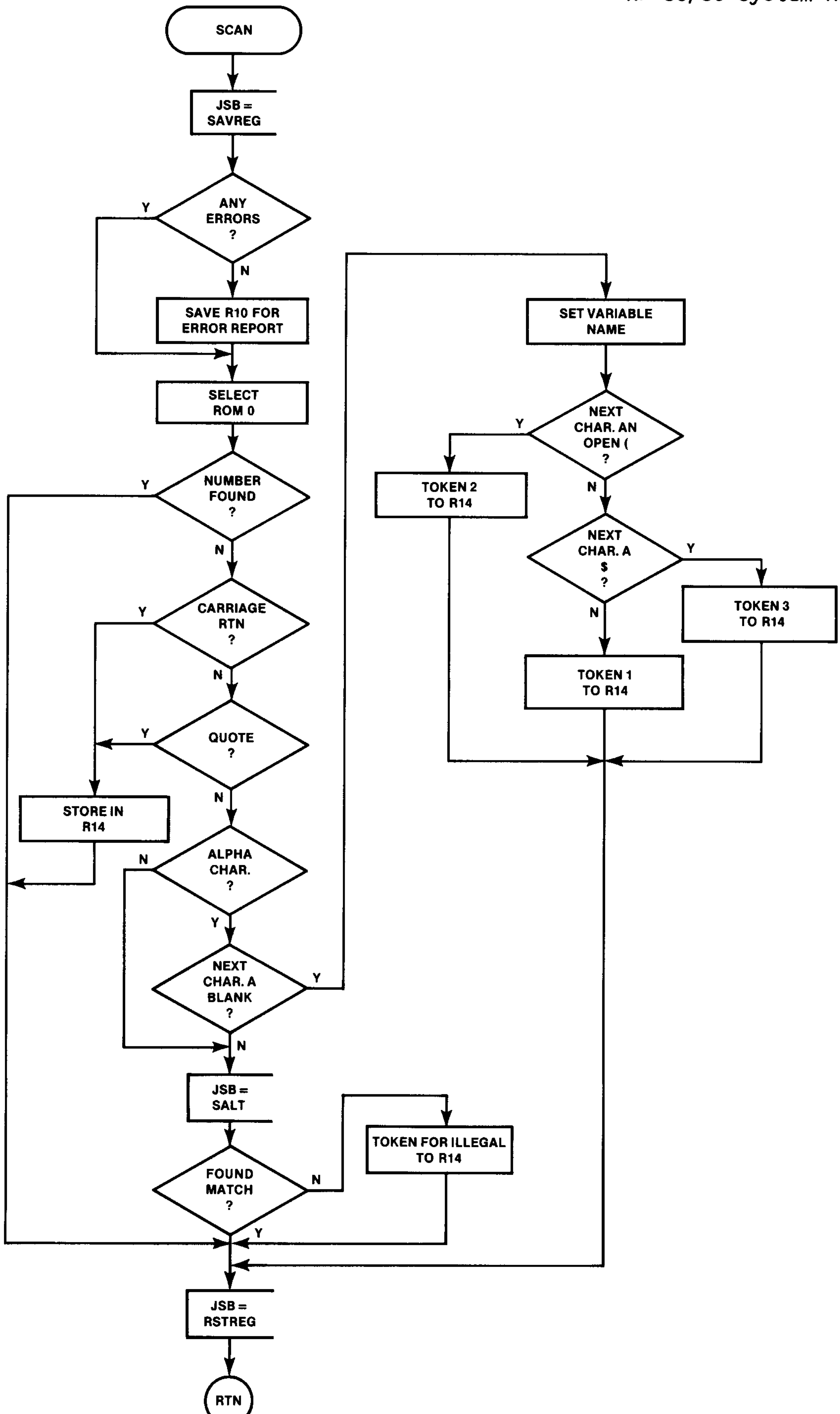
PARSING A PROGRAM LINE



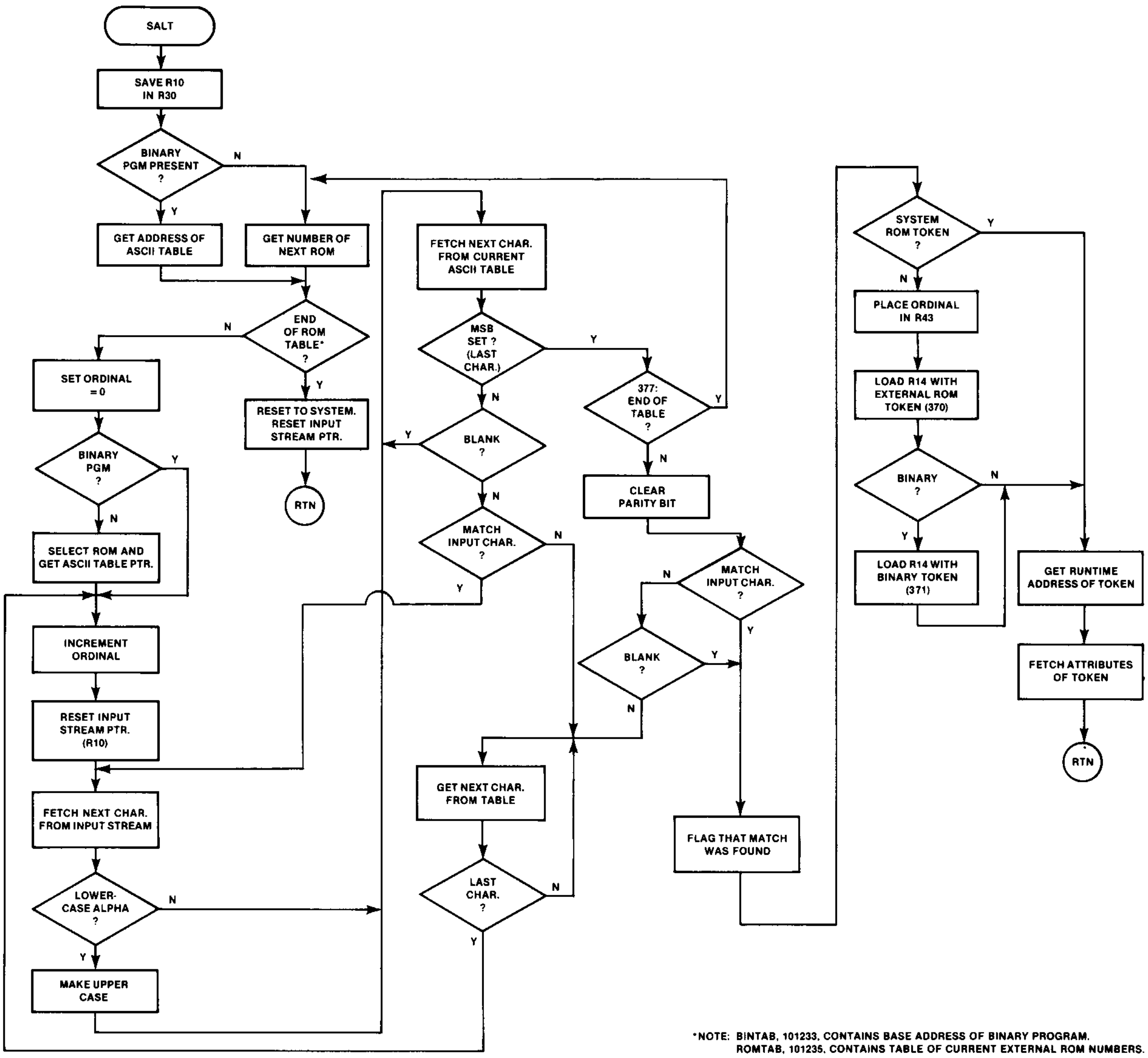
PARSING A CALCULATOR MODE STATEMENT



PARSIT ROUTINE



SCAN ROUTINE



*NOTE: BINTAB, 101233, CONTAINS BASE ADDRESS OF BINARY PROGRAM.
ROMTAB, 101235, CONTAINS TABLE OF CURRENT EXTERNAL ROM NUMBERS.

Main Parse Loop: In the main parse loop, if there is a line number, control passes to P.PARS, for parsing a program statement. If the statement has no line number, C.PARS parses a calculator mode statement.

Parsing a Program Line: P.PARS calls the PARSIT routine, then calls the EOL (end-of-line) and LINEDR (line editor) routines.

Parsing a Calculator Mode Statement: C.PARS calls the PARSIT routine, then checks for and processes any errors.

PARSIT Routine: The PARSIT routine calls another parse routine, SCAN.

SCAN Routine: The SCAN routine is always called in parsing. It is SCAN that places the next token in R14.

The SCAN routine finds the next token, or the next character if a token match cannot be found.

If the input is:	SCAN:
Digit	Places integer in R44 or floating-point on R12.
Period	Places floating-point quantity on R12.
Quotation mark symbol	Returns token 42 in R14 and does not execute GCHAR.
Anything in tables	Returns token.
Alpha not in tables	Returns variable type.
Other not in tables	Returns error token 17.
Blank	Skips the character.

SCAN FUNCTIONS

SCAN, in turn, calls the routine SALT.

SALT Routine: The SALT routine searches all ROM and binary program tables, one character at a time, looking for a keyword match.

PARSING IN BINARY PROGRAMS AND ROMS

A binary program or ROM gains control at parsetime when the system matches a keyword within that binary program or ROM. Once control is passed to the binary program or ROM, there are certain responsibilities of the parse routine before control is passed back to the calling location.

One responsibility is that SCAN must be called at entry to get the next token. SCAN may be called in one of three ways:
--Calling SCAN.
--Calling NUMVA+ (which calls SCAN first).
--Calling STREX+ (which also calls SCAN first).

When parsing is completed, SCAN must also be performed before returning to the system. However, most system parse routines (NUMVAL, STREXP, GETCMA, etc.) call SCAN before returning, so it is usually done for the user.

Another responsibility is that if a binary program is intended to be resident in an external ROM, the parse routines must ensure that ROM Ø is enabled when control is passed back to the system. This can be accomplished by executing GTO ROMRTN.

PARSE ROUTINE EXAMPLES

Here are some examples of parse routines for different functions:

<u>Statement With No Parameters</u>: e.g., BLOOPER

```
BLOPRS LDB 42, = 371                  Load binary program token marker.
       PUBD 42, +12                   Push it.
       PUBD 42, +12                   Push a garbage byte.
       PUBD 43, +12                   Push binary program token.
       JSB = SCAN                     Do a scan.
       RTN                            Return.
```

<u>Statement With One Parameter</u>: e.g., SLOOPER <u>numeric or string value</u>

```
SLOPRS PUBD 43, +6                    Save binary program token.
       JSB = NUMVA+                   Do a scan and try to get numeric.
```

```
       JEN GOTNUM               JIF found a numeric.
       JSB = STREXP             Try to get a string, then:
       JEN GOTNUM               JIF found a string.
       POBD 57, -6              Clean up RTN stack.
       JSB = ERROR+             Report error.
       BYT 81D                  Bad expression.
GOTNUM POBD 57, -6              Recover binary program token.
       LDB 55, = 371            Load binary program token marker.
       PUMD 55, +12             Push them.
       RTN                      Done.
```

Statement With More Than One Parameter (written for an external ROM): e.g., TROOPER numeric value, numeric value, string value

```
TROPRS PUBD 43, +6              Save ROM token.
       JSB ROMJSB   )
       DEF NUMVA+   }           Do a scan and get a numeric.
       BYT Ø        )
       JEN NUMOK                JIF got one.
ERR    POBD 57, -6              Clean up R6 stack.
       JSB = ERROR              Report error.
       BYT 88D                  Bad statement.
RTN    GTO ROMRTN               Ensure ROM Ø is reselected.
NUMOK  JSB = ROMJSB )
       DEF GETCMA   }           Demand a comma.
       BYT Ø        )
       JSB = ROMJSB )
       DEF NUMVAL   }           Try to get another numeric.
       BYT Ø        )
       JEZ ERR                  JIF not there to error.
       JSB = ROMJSB )
       DEF = GETCMA }           Demand another comma.
       BYT Ø        )
       JSB = ROMJSB
       DEF STREXP   )           Get a string expression.
       BYT Ø        }
       JEZ ERR      )           JIF not there.
```

```
        POBD 57, -6              Recover ROM token.
        LDB 56, = MYROM#         Load ROM number.
        LDB 55, = 370            Load ROM token marker.
        PUMD 55, +12             Push them all.
        JMP RTN                  Re-select ROM 0.
MYROM#  EQU 341
```

PARSE ROUTINES

System routines useful in parsing follow.

FUNCTION

Determines if next SCAN character is an alphabetic one (i.e., A-Z or a-z).

NAME	ALFA
ADDRESS	11775
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R20 = Current character being scanned	

OUTPUT CONDITIONS

E set to 1 if: A <= R20 <= Z (upper case)
or
a <= R20 <= z (lower case).
Otherwise E cleared to Ø.
If lower-case input, R20 is changed to upper-case for output; otherwise R20 is left unchanged.

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	-	U

COMMENTS — ROMJSB N

R20 contents may be changed if lower-case. No other registers are affected.
E used as output flag.
Mode changed to binary.

FUNCTION

Determines if next SCAN character is a digit (0-9; i.e., ASCII 60-71).

NAME	DIGIT
ADDRESS	12027
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R20 = Current character being scanned	

OUTPUT CONDITIONS

E set to 1 if 60_8 <= R20 <= 71_8; otherwise, E is cleared.

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS — ROMJSB N

Affects nothing but E-register.

NAME	DMNDCR
ADDRESS	15060
TYPE	Parse

FUNCTION

Checks R14 for either the carriage return character (15) or an exclamation point (233). Generates error if neither is found; returns if either is found.

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Current token	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Same current token	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	U	-	U

ROMJSB: Y

COMMENTS

This routine demands a carriage return or a remark after a line; if DMNDCR returns to the calling routine, a CR or a ! is guaranteed.

NAME	G$N
ADDRESS	14323
TYPE	Parse

FUNCTION

Parses one string followed by one number (e.g., BPLOT A$,1).

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Input token	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Next SCAN token	String expression tokens Numeric value tokens Token from R14 R12 → ------------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	**12**	**13**	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

ROMJSB: Y

COMMENTS

Calls STREX+ and GETPA-. (Similar to GETPAR.)

FUNCTION

Parses one string followed by 1 or 2 numeric parameters (e.g., CREATE A$, n [,m]).

NAME	G$N+NN
ADDRESS	14421
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Input token	
OUTPUT CONDITIONS	R14 = Next SCAN token	String expression tokens 1 or 2 numeric value tokens Token from R14 R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	-	-
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67	-	
70	71	72	73	74	75	76	77		

COMMENTS ROMJSB Y

Calls another routine which demands 1 or 2 numerics. (R34 = 1 or 2.)

FUNCTION

Gets Ø, 1 or 2 numeric parameters.

NAME	GØ12N
ADDRESS	14465
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	Normal parse input, i.e.: R10 = Input buffer pointer R14 = Next token R20 = Next character in input buffer	Stack output pointer
OUTPUT CONDITIONS	Normal parse output, i.e.: R14 = Next token R40-47 = Current parse information	Results of successful parse R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	14	12
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS ROMJSB Y

Parses Ø, 1 or 2 line numbers separated by "," (1 <= line number <= 9999). Calls SEQNO+ for line number. Error 90 if line number outside specified range. Error 91 if "," not followed by another line number.

FUNCTION

Same as GØ12N, except gets Ø or 1 numeric parameters.

NAME GØ1N
ADDRESS 14504
TYPE Parse

INPUT CONDITIONS

REGISTER CONTENTS

Normal parse input (see SCAN)

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

R12 → Results of successful parse

CPU CHANGES

(shaded registers shown in bold)

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	14	12	U

COMMENTS

ROMJSB Y

See GØ12N.

FUNCTION

Same as GØ12N, except gets Ø or 2 numeric parameters.

NAME GØOR2N
ADDRESS 14522
TYPE Parse

INPUT CONDITIONS

REGISTER CONTENTS

Normal parse input (See SCAN)

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

R12 → Results of successful parse

CPU CHANGES

(shaded registers shown in bold)

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	U	34	U	U

COMMENTS

ROMJSB Y

Error 91 if only one parameter. Calls NUMVA+ to get numeric value.

FUNCTION	NAME	G12OR4
Same as GØ12N except gets 1, 2 or 4 numeric parameters.	ADDRESS	14550
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token	
OUTPUT CONDITIONS	R14 = Next SCAN token R20 = Next character (Set by SCAN) R34 = Number of parameters found (Error exit if ≠ 1, 2 or 4) R35 = Ø R40 = Set by SCAN	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

ROMJSB: Y

COMMENTS

Calls GETPA?.
E≠Ø if parameters found.

FUNCTION	NAME	G1OR2N
Same as GØ12N, except gets 1 or 2 numeric parameters.	ADDRESS	14537
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Current token	
OUTPUT CONDITIONS	R14 = New current token	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	34	-	-

ROMJSB: Y

COMMENTS

Calls GETPA?.
Aborts through ERROR+ (91) if error in finding parameters. Aborts through another routine if too many parameters.

FUNCTION

Fetches next character (usually from input buffer) addressed by R10 pointer. GCHAR skips blanks, and it increments R10 unless the character is a carriage return.

NAME	GCHAR
ADDRESS	11755
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R10 = Pointer to character	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R10 = Pointer to following character (unless present character was a carriage return) R20 = Character popped from R10	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	-	-	-

COMMENTS ROMJSB Y

Performs SAD at entry, PAD at exit.

FUNCTION

Demands one string; also gets one numeric if present. Used to parse a statement with one string, and that may have one numeric parameter. Generates error if no string found.

NAME	GET$N?
ADDRESS	14560
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
Normal parse inputs (See SCAN)	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Parse results R12 → -------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	14	-	-

COMMENTS ROMJSB Y

Parses: string expression or string expression, followed by 1 or 2 line numbers.
Possible errors: 90 if line number out of range. 91 if , not followed by another line number.

FUNCTION

Looks for the symbol) in R14 (usually following a call to SCAN). If) is found, calls SCAN and clears E; otherwise, aborts through ERROR+ with error 80D.

NAME	GET)
ADDRESS	13365
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Current input buffer symbol	
OUTPUT CONDITIONS	R14 = New current symbol, if successful	

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS ROMJSB Y

Expects E cleared (≠-1) at entry.
At exit, E signals whether) was found:
E=E+1 means) was found.
E=Ø means) was not found.

FUNCTION

Demands a string expression and processes it.

NAME	GET1$
ADDRESS	14455
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R10 = Pointer to input stream R14 = Current token	R12 → Output stack
OUTPUT CONDITIONS	R14 = Next token R40-47 = Set by SCAN	

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	14	12	U

COMMENTS ROMJSB Y

Calls STREX+.
Returns an error if no string is found.

FUNCTION	NAME	GET1N
Gets one numeric parameter, and pushes onto R12 the corresponding numeric value token and the token in R14.	ADDRESS	14437
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token	
OUTPUT CONDITIONS	R14 = Next SCAN token R20 = Next character (Set by SCAN) R34 = Number of parameters found (Error if ≠ 1) R35 = 1 (Set by GETPAR) R40 = Set by SCAN	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

COMMENTS — ROMJSB: Y

Sets R35 = 1, then calls GETPAR. E≠Ø if found.

FUNCTION	NAME	GET2N
Gets two numeric parameters, and pushes onto R12 the corresponding numeric value tokens and the token in R14.	ADDRESS	14407
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Current SCAN token	
OUTPUT CONDITIONS	R14 = Next SCAN token R20 = Next character (Set by SCAN) R34 = Numer of parameters found (Generates error if ≠ 2) R35 = 2 R40 = Set by SCAN	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

COMMENTS — ROMJSB: Y

E≠Ø if found.
GET2N jumps to GETPAR.

FUNCTION

Gets four numeric parameters and pushes onto R12 the corresponding numeric value tokens and the token in R14.

NAME	GET4N
ADDRESS	14414
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS

R14 = Current SCAN token

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R14 = Next SCAN token
R20 = Next character (Set by SCAN)
R34 = Number of parameters found (Generates error if ≠ 4)
R35 = 4
R40 = Set by SCAN

R12 STACK CONTENTS

Numeric value tokens
Token from R14
R12 → --------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	**12**	**13**	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

COMMENTS

ROMJSB: Y

E≠Ø if found.
GET4N jumps to GETPAR.

FUNCTION

Demands a comma as the next SCAN token. Sets E≠Ø if found; otherwise, returns an error.

NAME	GETCMA
ADDRESS	13414
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS

R14 = SCAN token
R40 = Set by SCAN

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R14 = Next token
R40 = Set by SCAN

R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS

ROMJSB: Y

E≠Ø if comma is found.

FUNCTION	NAME	GETCM?
Checks for a comma. Sets E≠0 if found.	ADDRESS	13425
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token R40 = Set by SCAN	
OUTPUT CONDITIONS	R14 = Next token, if SCAN token was a comma R40 = Set by SCAN	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS — ROMJSB Y

E≠0 if comma found.

FUNCTION	NAME	GETPA?
Gets an arbitrary number of numeric parameters. (Same as GETPAR except R35 is set to zero.)	ADDRESS	14516
	TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Input token R35 = 0 (Then GETPAR is called)	
OUTPUT CONDITIONS	R14 = Next SCAN token R34 = Number of numeric parameters found	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	34	-	-

COMMENTS — ROMJSB Y

Same as GETPAR with R35 = 0. Calls NUMVA+.

FUNCTION	
GETPAR gets as many numeric parameters as it can. If at entry R35 = Ø, any number is acceptable. If R35 ≠ Ø, the number fetched must equal that in R35. GETPAR pushes the input token.	NAME GETPAR ADDRESS 14342 TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Input token R35 = Ø (Any number of parameters) or R35 ≠ Ø (R35 = Number of parameters)	
OUTPUT CONDITIONS	R34 = Number of parameters found. If R35 ≠ Ø, then R34 = R35; otherwise, an error is returned.	Numeric value tokens Token from R14 R12 → --------------------

CPU CHANGES (shaded registers shown in bold)

0	1	**2**	**3**	4	5	6	7
10	11	**12**	**13**	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

COMMENTS — ROMJSB Y

Calls NUMVA+.

FUNCTION	
Tries to get an integer of up to 14_{10} digits from input buffer. Used in applications such as sequence numbers, where it is desired to ignore decimal points and exponents.	NAME INTEGR ADDRESS 11447 TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R10 = Input buffer pointer (Next character) R20 = Current character from buffer	
OUTPUT CONDITIONS	R10 = Next character in buffer after number R20 = First non-digit character R36 = Exponent of integer in R40 R40 = Digits of number found	

CPU CHANGES (shaded registers shown in bold)

0	1	2	3	4	5	6	7
10	**11**	12	13	14	15	16	17
20	21	**22**	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	22	U	U

COMMENTS — ROMJSB Y

No SCAN is necessary before INTEGR is called.
E=Ø if no number found, E=1 if found.
On return, R40 contains right-justified number if R36 = 15C; otherwise R40 contains integer with exponent of R36-15C.

FUNCTION

Same as NARREF, except that it performs a SCAN first.

NAME	NARRE+
ADDRESS	13376
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Next token R40-47 = As per SCAN outputs	2 (Fetch array token) R44 } Name R45 } R12 → --------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	1	14	36	U

COMMENTS

ROMJSB Y

Calls SCAN at both entry and exit.

FUNCTION

Parses a simple numeric variable reference as an array reference (i.e., MATA=Ø).

NAME	NARREF
ADDRESS	13402
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Current token (Should be a 1)	R12 → --------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Next token R40-47 = As per SCAN outputs	2 (Fetch array token) R44 } Name R45 } R12 → --------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	1	14	36	U

COMMENTS

ROMJSB Y

Calls SCAN before returning.

FUNCTION

Pushes integer or floating point number onto the R12 stack and calls SCAN.

NAME	NUMCON
ADDRESS	13466
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS

R14 = Token from SCAN (4 if floating point, 32 if integer)
R40 = Set by SCAN

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R14 = Next token from SCAN
R40 = Set by SCAN

R12 STACK CONTENTS

Integer or floating point number
R12 → --------------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47	-	-
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	-	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

Must SCAN before calling this routine.
Routine SCANs before exit.
At exit, E≠Ø if number found.

FUNCTION

Same as SCAN routine followed by NUMVAL routine.

NAME	NUMVA+
ADDRESS	12407
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17		
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

See NUMVAL for conditions and changes.

FUNCTION

Pushes a numeric value onto the R12 stack and calls SCAN. Sets E≠Ø if numeric found; otherwise sets E=Ø and restores input buffer.

NAME	NUMVAL
ADDRESS	12412
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token R40 = Set by SCAN	
OUTPUT CONDITIONS	R14 = Next SCAN token R20 = Set by SCAN R40 = Next character (Set by SCAN)	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	-	-
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	-	

COMMENTS

ROMJSB Y

E≠Ø if numeric found.
Calls SCAN at exit.

FUNCTION

Pushes the token in R14 onto the R12 stack and calls SCAN.

NAME	PUSH1A
ADDRESS	14244
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Token	
OUTPUT CONDITIONS	R14 = Next token (Set by SCAN) R20 = Next character (Set by SCAN) R40 = Set by SCAN	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	1
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	14	36
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	-	

COMMENTS

ROMJSB Y

Calls SCAN before exit.
Sets E=1.

FUNCTION

Pushes an integer onto the R12 stack (at parse time).

NAME	PUSH32
ADDRESS	14277
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = 32 (Integer token) R44-46 = BCD integer	
OUTPUT CONDITIONS	R14 = Next token R40-47 = Set by SCAN	32 (Integer token) BCD integer value from R44-46 R12 → ----------------------------

CPU CHANGES (changed registers shown in bold)

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	1	14	36	U

COMMENTS

ROMJSB: Y

Sets E=1.
Calls SCAN at exit.

FUNCTION

Pushes the token in R14 onto the R12 stack; then pushes the variable name in R44-45 onto the stack and calls SCAN.

NAME	PUSH45
ADDRESS	14266
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Token R44-45 = Variable name	
OUTPUT CONDITIONS	R14 = Next token (Set by SCAN) R20 = Next character (Set by SCAN) R40 = Set by SCAN	

CPU CHANGES (changed registers shown in bold)

0	1	**2**	**3**	4	5	6	7
10	11	**12**	**13**	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	1	14	36	-

COMMENTS

ROMJSB: Y

Jumps to another routine, which calls SCAN.
Sets E=1.

FUNCTION

Parses a simple numeric variable or a numeric array reference.

NAME REFNUM
ADDRESS 17025
TYPE Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = 1 if simple numeric variable reference. = 2 if array reference. Otherwise, exit.	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Next SCAN token	21 Variable name OR 22 Array name Parsed subscript

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: - E: U DRP: - ARP: - STATUS: -

COMMENTS

ROMJSB Y

E=Ø at entry.
E≠Ø at exit (if found).

FUNCTION

Scans the input buffer and returns the next token.

NAME SCAN
ADDRESS 11262
TYPE Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R10 = Input buffer pointer R20 = Next character in input buffer	R12 = Output stack pointer

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R10 = Input buffer pointer R14 = Next token R20 = Next character R40 = First character searched R41-42 = ROM#, binary program address, or Ø R43 = ROM token or variable type R44-46 = Name of var. or int., or sec. att. R47 = Class	R12 = Output stack pointer

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: - E: 0 DRP: 14 ARP: 36 STATUS: U

COMMENTS

ROMJSB Y

E=Ø at exit.

FUNCTION

Gets next character (through GCHAR) and executes SCAN.

NAME	SCAN+
ADDRESS	11257
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB Y

See SCAN for conditions and changes.

FUNCTION

Pushes current token onto R12 stack and looks for valid sequence (line) number. Pushes line number if found.

NAME	SEQNO+
ADDRESS	17454
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Current token	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = New current token	Current token Sequence number (2-byte integer. Present only if found.) R12 → ----------------------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS

ROMJSB Y

Expects E cleared (≠-1) at entry.
Calls SEQNO (which calls SCAN) to get an integer.
Generates error if sequence number = Ø, or if number > 9999.
Sets E=E+1 if sequence number found; sets E=Ø if number not found.

FUNCTION

Scans for sequence (line) number, and pushes the number onto the R12 stack.

NAME	SEQNO
ADDRESS	17457
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
If no sequence number found: R14 = New current token	If sequence number found: R12 stack = Sequence number (2-byte integer)

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS ROMJSB Y

Expects E cleared (≠-1) at entry.
Calls SEQNO (which calls SCAN) to get an integer.
Generates an error if sequence number = Ø, or if sequence number > 9999.
Sets E=E+1 if sequence number found; sets E=Ø if no sequence number found.

FUNCTION

Pushes an integer (R44-46) at parse time if R14 contains integer token (32).

NAME	SMLINT
ADDRESS	13474
TYPE	Parse

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R14 = Current token	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
If R14 = 32 at entry: R14 = Next token R40-47 = Set by SCAN Otherwise, registers unchanged	If R14 = 32 at entry: E=1 and stack contents are: 32 (Integer token) R44-46 Value R12 → -------------------- Otherwise, R12 unchanged and E=Ø

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	U	U	U

COMMENTS ROMJSB Y

Calls SCAN at exit.
If R14≠32 at entry, then at exit E=Ø, DRP=14, and nothing else is changed.

FUNCTION

Pushes a quoted string onto the R12 stack, then calls SCAN.

NAME STRCON
ADDRESS 14036
TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Token (Must be quote) R40 = Set by SCAN	
OUTPUT CONDITIONS	R14 = Next SCAN token R40 = Set by SCAN	5 Number of bytes in string String R12 → ------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	**12**	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	**46**	47
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
U	U
DRP	**ARP**
U	U
STATUS	
U	

COMMENTS ROMJSB Y

Must SCAN before entry to this routine. Routine SCANs
before exit.
E≠0 if quoted string found.

FUNCTION

Same as SCAN followed by STREXP.

NAME STREX+
ADDRESS 13623
TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
DRP	**ARP**
STATUS	

COMMENTS ROMJSB Y

See STREXP for conditions and changes.

FUNCTION

Pushes a string expression onto the R12 stack. E≠Ø if found.

NAME	STREXP
ADDRESS	13626
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token R40 = Set by SCAN	
OUTPUT CONDITIONS	R14 = Next SCAN token R40 = Set by SCAN	

CPU CHANGES: 2, 3, 20, 40–47 (shaded); DCM: –; E: U; DRP: –; ARP: –; STATUS: –

COMMENTS — ROMJSB: Y

Must SCAN before calling this routine. The routine SCANs before exit.
E≠Ø if string expresion is found.

FUNCTION

Pushes a string variable or a substring reference onto the R12 stack, then calls SCAN.

NAME	STRREF
ADDRESS	13753
TYPE	Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = SCAN token R40 = Set by SCAN	
OUTPUT CONDITIONS	R14 = Next SCAN token R40 = Set by SCAN	

CPU CHANGES: 14, 15, 20, 36, 40–47 (shaded); DCM: –; E: U; DRP: –; ARP: –; STATUS: –

COMMENTS — ROMJSB: Y

Must SCAN before calling this routine. This routine SCANs before exit.
E≠Ø if found.

FUNCTION	
Gets Ø or 1 numeric parameter and pushes token from R14 onto R12 stack.	NAME TRY1N ADDRESS 14566 TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R14 = Input token	
OUTPUT CONDITIONS	R14 = Next SCAN token R34 = Ø or 1 (2 or more produces error)	Ø or 1 numeric value tokens Token from R14 R12 → ----------------------------

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	**12**	**13**	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	U	34	-	-

COMMENTS — ROMJSB Y

Calls GETPA?, then demands R34 < 2.

FUNCTION	
Pushes an unquoted string onto the R12 stack, then calls SCAN. E≠Ø if unquoted string found.	NAME UNQUOT ADDRESS 14212 TYPE Parse

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R20 = First character of string	
OUTPUT CONDITIONS	R14 = Next SCAN token R40 = Set by SCAN	

CPU CHANGES

0	1	**2**	3	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	-	-	-

COMMENTS — ROMJSB Y

E≠Ø if found.
This routine calls SCAN before exit.

RUNTIME AND RUNTIME ROUTINES

RUNTIME CONVENTIONS

System routines used at runtime include primarily mathematics routines and system functions. In general, math routines always expect BCD mode at entry. System functions expect the argument(s) on the R12 stack when the routine is called, and leave the result on the R12 stack when completed.

CPU registers used during runtime include, but are by no means limited to, the ones shown here.

Register	Runtime Use
R12	Operating stack pointer.
R16	Contains CSTAT.
R17	Contains XCOM.

RUNTIME ROUTINES

System routines useful at runtime follow.

FUNCTION

Returns the absolute value of the argument.

NAME	ABS5
ADDRESS	53731
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Absolute value (8 bytes) R12 → ------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E
D	U
DRP	**ARP**
40	12
STATUS	
U	

COMMENTS

ROMJSB: N

FUNCTION

Adds two numbers (X+Y).

NAME	ADDROI
ADDRESS	52150
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-value (8 bytes) Y-value (8 bytes) R12 → ----------------
OUTPUT CONDITIONS	R40 = Copy of result	X+Y value (8 bytes) R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E
D	U
DRP	**ARP**
40	12
STATUS	
U	

COMMENTS

ROMJSB: N

FUNCTION

Returns arctangent of Y/X (i.e., ATN2) in proper quadrant.

NAME	ATN2.
ADDRESS	76455
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Y-value (8 bytes) X-value (8 bytes) R12 → -----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	ATN2 value (8 bytes) R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Executes the BEEP statement.

NAME	BEEP.
ADDRESS	6737
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Parameter #1 (8 bytes) (optional) Parameter #2 (8 bytes) (optional) R12 → ----------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ----------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

If no parameters are on stack at entry (i.e., TOS = R12), a standard beep is executed.

FUNCTION	NAME	CEIL1Ø
Returns the smallest integer > = x.	ADDRESS	53615
	TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-value (8 bytes) R12 → ----------------
OUTPUT CONDITIONS	R40 = Copy of CEIL result	CEIL result (8 bytes) R12 → --------------------

CPU CHANGES (shaded registers in bold)

0	1	**2**	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION	NAME	CHSROI
Changes the sign of a number.	ADDRESS	52075
	TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Number (8 bytes) R12 → ---------------
OUTPUT CONDITIONS		-Number (8 bytes) R12 → ----------------

CPU CHANGES (shaded registers in bold)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

Requires BCD mode at entry.

ROMJSB N

FUNCTION

Prints a string to the print buffer or the display buffer.
(Same as PRINT A$, in BASIC.)

NAME	COMMA$
ADDRESS	70634
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Length of string (2 bytes) Address of string (2 bytes) R12 → --------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS — ROMJSB: Y

DISP. or PRINT. <u>must</u> be called prior to calling COMMA$ to set up the select code and buffer pointers.

FUNCTION

Prints a number to the print buffer or the display buffer.
(Same as PRINT 5, in BASIC.)

NAME	COMMA.
ADDRESS	70756
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Number (8 bytes) R12 → ----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ----------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS — ROMJSB: Y

DISP. or PRINT. <u>must</u> be called prior to calling COMMA. to set up select code and buffer pointers.

FUNCTION

Concatenates two strings.

NAME	CONCA.
ADDRESS	75005
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		A$ length (2 bytes) A$ address (2 bytes) B$ length (2 bytes) B$ address (2 bytes) R12 → -------------------
OUTPUT CONDITIONS		A$ and B$ length A$ and B$ address R12 → ----------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Returns cosine of argument.

NAME	COS10
ADDRESS	53556
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (real or integer #) R12 → ----------------------------
OUTPUT CONDITIONS	R40 = Copy of result	Answer (real #) R12 → ---------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns the cotangent of the argument.

NAME	COT1Ø
ADDRESS	53536
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Argument (8 bytes) R12 → ------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of cotangent result	Cotangent (8 bytes) R12 → -------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns cosecant of argument.

NAME	CSEC1Ø
ADDRESS	53503
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Argument (8 bytes) R12 → ------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of cosecant result	Cosecant (8 bytes) R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns current date.

NAME	DATE.
ADDRESS	37673
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Date (8 bytes) R12 → --------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
–	–	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Turns defaults on.

NAME	DEFA+.
ADDRESS	61505
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
–	–	36	-	U

COMMENTS

ROMJSB Y

FUNCTION

Turns defaults off.

NAME	DEFA-
ADDRESS	61513
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	36	-	U

COMMENTS

ROMJSB Y

FUNCTION

Sets computer to degrees mode for trogonometric operations.

NAME	DEG.
ADDRESS	61736
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	36	-	U

COMMENTS

ROMJSB Y

FUNCTION

Converts angle in radians to degrees.

NAME DEG1Ø
ADDRESS 54142
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Angle in radians (8 bytes) R12 → --------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of result	Angle in degrees (8 bytes) R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Sets SCTEMP and PRINT pointers to CRT IS device.

NAME DISP.
ADDRESS 70046
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	40	-	U

COMMENTS

ROMJSB Y

FUNCTION

Divides Y into X.

NAME	DIV2
ADDRESS	51641
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-value (8 bytes) Y-value (8 bytes) R12 → -----------------
OUTPUT CONDITIONS	R40 = Copy of result	X/Y value (8 bytes) R12 → -------------------

CPU CHANGES: DCM D; E U; DRP 40; ARP 12; STATUS U

COMMENTS

ROMJSB N

FUNCTION

Returns the smallest positive number (1E-499) the computer is capable of handling.

NAME	EPS10
ADDRESS	54126
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS	R40 = Copy of smallest number	Smallest number (8 bytes) R12 → -------------------------

CPU CHANGES: DCM D; E U; DRP 50; ARP 12; STATUS U

COMMENTS

ROMJSB N

FUNCTION

Divides two real numbers.

NAME	DIV1Ø
ADDRESS	51644
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R50 = Real #A (Numerator)
R40 = Real #B (Denominator)

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R40-47 = Real rounded result (Copy)

R12 STACK CONTENTS

Real rounded result (A/B)
R12 → -------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

Not listed in global file. Same as DIV2, except DIV2 expects two real or integer numbers on the R12 stack.

FUNCTION

NAME	
ADDRESS	
TYPE	

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB

FUNCTION

Compares two numbers for equality. (#1 = #2.)

NAME EQ.
ADDRESS 62173
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	#1 Value (8 bytes) #2 Value (8 bytes) R12 → -----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	True/false value (8 bytes) R12 → -------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	**36**	**37**	40	12
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	STATUS	
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**	U	
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

FUNCTION

Compares two strings for equality.

NAME EQ$.
ADDRESS 3006
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	String 1 length (2 bytes) String 1 address (2 bytes) String 2 length (2 bytes) String 2 address (2 bytes) R12 → -------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	True/false value (8 bytes) R12 → -------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	D	U
20	21	22	23	24	25	26	27	DRP	ARP
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	40	12
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	STATUS	
50	**51**	52	53	54	55	56	57	U	
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Sets ERRORS, ERRN, ERRL and error flag in R17.

NAME	ERROR
ADDRESS	6615
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS ROMJSB N

ERROR must be called with the following code:

```
JSB=ERROR          Call to ERROR.
BYT ___            Error number.
```

FUNCTION

Sets ERRORS, ERRN, ERRL and error bit in R17, then pops one return address off of R6 before returning.

NAME	ERROR+
ADDRESS	6611
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS ROMJSB N

ERROR+ must be called with the following code:

```
JSB=ERROR+         Call to ERROR+.
BYT ___            Error number.
```

FUNCTION

Returns e^X.

NAME	EXP5
ADDRESS	52377
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-value (8 bytes) R12 → ----------------
OUTPUT CONDITIONS		e^X result (8 bytes) R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	21	22	23	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns the fractional portion of the argument.

NAME	FP5
ADDRESS	54071
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → -----------------
OUTPUT CONDITIONS	R40 = Copy of result	Result (8 bytes) R12 → ---------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Compares two numbers for the condition: #1 >= #2.

NAME	GEQ.
ADDRESS	62304
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
      #1 value (8 bytes)
      #2 value (8 bytes)
R12 → ------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
      True/false value (8 bytes)
R12 → --------------------------
```

CPU CHANGES

DCM	E	DRP	ARP	STATUS
U	U	40	12	U

COMMENTS

ROMJSB: Y

FUNCTION

Compares two strings for the condition: string 1 >= string 2.

NAME	GEQ$.
ADDRESS	3111
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
      String 1 length (2 bytes)
      String 1 address (2 bytes)
      String 2 length (2 bytes)
      String 2 address (2 bytes)
R12 → --------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
      True/false value (8 bytes)
R12 → --------------------------
```

CPU CHANGES

DCM	E	DRP	ARP	STATUS
D		40	12	

COMMENTS

ROMJSB: N

FUNCTION

Sets computer to grads mode for trigonometric operations.

NAME	GRAD.
ADDRESS	61753
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
–	–	36	–	U

COMMENTS

ROMJSB Y

FUNCTION

Compares two numbers for the condition: #1 > #2.

NAME	GR.
ADDRESS	62255
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      #1 value (8 bytes)
      #2 value (8 bytes)
R12 → ------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      True/false value (8 bytes)
R12 → --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	40	12	U

COMMENTS

ROMJSB Y

FUNCTION

Compares two strings for the condition: string 1 > string 2.

NAME	GR$.
ADDRESS	3036
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		String 1 length (2 bytes) String 1 address (2 bytes) String 2 length (2 bytes) String 2 address (2 bytes) R12 → -------------------------
OUTPUT CONDITIONS		True/false value (8 bytes) R12 → -------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB: N

FUNCTION

Returns inverse consine (arccosine) of argument.

NAME	ICOS
ADDRESS	76552
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Arc cosine (8 bytes) R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB: Y

FUNCTION

Returns largest number (9.99999999999E499) that can be handled by the computer.

NAME INF1Ø
ADDRESS 53524
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → -------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of largest number	9.99999999999E499 (8 bytes) R12 → --------------------------

CPU CHANGES (changed registers: 0, 1, 40–47)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	Ø	6	U

COMMENTS

ROMJSB N

FUNCTION

Returns the FLOOR of X. (Largest integer < = X.)

NAME INT5
ADDRESS 53776
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-value (8 bytes) R12 → ----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of result	Result (8 bytes) R12 → ---------------

CPU CHANGES (changed registers: 2, 32–37, 40–47, 50–57, 60–67)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION	
Performs integer division: divides two numbers and returns an integer result.	NAME INTDIV ADDRESS 54005 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		#A (8 bytes) #B (8 bytes) R12 → ------------
OUTPUT CONDITIONS		A\B result (8 bytes) R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	12	U

COMMENTS

ROMJSB N

FUNCTION	
Returns integer portion of the argument.	NAME IP5 ADDRESS 54174 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Result (8 bytes) R12 → ----------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	12	U

COMMENTS

E=Ø FOR TAGGED INTEGER
E≠Ø FOR FLOATING POINT

ROMJSB N

FUNCTION	
Returns inverse sine (arcsine) of argument.	NAME ISIN ADDRESS 76542 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Arcsine (8 bytes) R12 → -----------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION	
Returns inverse tangent (arctangent) of argument.	NAME ITAN ADDRESS 76562 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Arctangent result (8 bytes) R12 → ----------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Compares two numbers for the condition: #1 <= #2.

NAME	LEQ.
ADDRESS	62232
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       #1 Value (8 bytes)
       #2 Value (8 bytes)
R12 →  -------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       True/false value (8 bytes)
R12 →  --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	40	12	U

COMMENTS

ROMJSB Y

FUNCTION

Compares one string to a second string for the case:
string 1 <= string 2.

NAME	LEQ$.
ADDRESS	3100
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       String 1 length (2 bytes)
       String 1 address (2 bytes)
       String 2 length (2 bytes)
       String 2 address (2 bytes)
R12 →  --------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       True/false value (8 bytes)
R12 →  --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	52	53	54	55	56	57
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns LN(X).

NAME	LN5
ADDRESS	51551
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-value (8 bytes) R12 → -----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of result	LN(X) result (8 bytes) R12 → ----------------------

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7	D	U
10	11	12	13	14	15	16	17	**DRP**	**ARP**
20	21	22	23	24	25	26	27	40	12
30	31	32	33	34	35	36	37	**STATUS**	
40	41	42	43	44	45	46	47	U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Returns LOG_{10} (X)

NAME	LOGT5
ADDRESS	51720
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-value (8 bytes) R12 → -----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of result	LOG_{10} (X) result (8 bytes) R12 → --------------------------

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7	BCD	U
10	11	12	13	14	15	16	17	**DRP**	**ARP**
20	21	22	23	24	25	26	27	40	12
30	31	32	33	34	35	36	37	**STATUS**	
40	41	42	43	44	45	46	47	U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Compares two numbers for the case: #1 < #2.

NAME	LT.
ADDRESS	62213
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      #1 Value (8 bytes)
      #2 Value (8 bytes)
R12 → -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      True/false value (8 bytes)
R12 → --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	40	12	U

COMMENTS

ROMJSB Y

FUNCTION

Compares two strings for the condition: string 1 < string 2.

NAME	LT$.
ADDRESS	3057
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      String 1 length (2 bytes)
      String 1 address (2 bytes)
      String 2 length (2 bytes)
      String 2 address (2 bytes)
R12 → --------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      True/false value (8 bytes)
R12 → --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	52	53	54	55	56	57
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns the larger of two values.

NAME	MAX1Ø
ADDRESS	55364
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Value #1 (8 bytes) Value #2 (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Larger value (8 bytes) R12 → ----------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns the remainder (modulo) of division.

NAME	MOD1Ø
ADDRESS	51744
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		A-value (8 bytes) B-value (8 bytes) R12 → -----------------
OUTPUT CONDITIONS		A MOD B result (8 bytes) R12 → ------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION	
Returns the smaller of two values.	NAME MIN1Ø ADDRESS 55345 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Value #1 (8 bytes) Value #2 (8 bytes) R12 → ------------------
OUTPUT CONDITIONS		Smaller value (8 bytes) R12 → -----------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	12	U

COMMENTS

ROMJSB N

FUNCTION	
	NAME ADDRESS TYPE

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB

FUNCTION	
Multiplies two real numbers.	NAME MPY1Ø ADDRESS 52562 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS

INPUT CONDITIONS	R40 = #A (Real)
	R50 = #B (Real)	

OUTPUT CONDITIONS	R40 = Copy of result	Real result (A*B) R12 → -----------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS — ROMJSB N

Not in global file. Same as MPYROI, except MPYROI expects two real or integer numbers on the R12 stack at entry.

FUNCTION	
	NAME ADDRESS TYPE

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS — ROMJSB

FUNCTION

Multiplies two numbers.

NAME MPYROI
ADDRESS 52722
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      X-value (8 bytes)
      Y-value (8 bytes)
R12 → -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R40 = Copy of result

R12 STACK CONTENTS

```
      X * Y result (8 bytes)
R12 → ----------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Turns off one of the system timers.

NAME OFTIM.
ADDRESS 66211
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      Timer number (8 bytes)
R12 → ----------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
R12 → ----------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	**21**	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	**37**
40	41	42	43	44	45	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	55	46	U

COMMENTS

ROMJSB Y

FUNCTION

Pushes value of pi onto R12 stack as a real number.

NAME	PI1Ø
ADDRESS	53577
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		R12 → --------------------
OUTPUT CONDITIONS	R40 = Copy of pi (as real number)	Pi (as real number) R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	-
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37		
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	40	12
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

Pi = 31, 41C, 59C, 26C, 53C, 59C, 0, 0 (BCD)

Pi = 61, 101, 131, 46, 123, 131, 0, 0 (octal)

FUNCTION

Finds the character position in string A of the first occurrence of string B.

NAME	POS.
ADDRESS	3435
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Length of string A (2 bytes) Address of string A (2 bytes) Length of string B (2 bytes) Address of string B (2 bytes) R12 → ----------------------------
OUTPUT CONDITIONS		Position (8 bytes) R12 → -----------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	**24**	**25**	**26**	**27**	DRP	ARP
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**		
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	U	U
50	**51**	**52**	53	54	55	**56**	**57**	STATUS	
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

FUNCTION

Sets SCTEMP and PRINT pointers to PRINTER IS device.

NAME	PRINT.
ADDRESS	70067
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES (shaded: 40–47, 50–57)

DCM	E	DRP	ARP	STATUS
-	-	40	-	U

COMMENTS

ROMJSB: Y

FUNCTION

Dumps either the print buffer or the display buffer.

NAME	PRLINE
ADDRESS	70402
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES (shaded: 0–3, 14–15, 20–77)

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB: Y

DISP. or PRINT. <u>must</u> be called to set up select code and buffer pointers before calling PRLINE.

FUNCTION

Prints a string to a tape buffer.

NAME	PRNT#$
ADDRESS	30577
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R44-45 = Length of string R46-47 = Address of string	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

DCM	E	DRP	ARP	STATUS
BIN	U	U	U	U

COMMENTS ROMJSB Y

Before calling PRNT#$ a buffer must have been assigned and PRINT#. called.

FUNCTION

Prints a number to a tape buffer.

NAME	PRNT#N
ADDRESS	31022
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Number to be printed	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS ROMJSB Y

Before calling PRNT#N, a buffer must have been assigned and PRINT#. called.

FUNCTION

Sets the computer to radians mode for trigonometric operations.

NAME	RAD.
ADDRESS	61746
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES (shaded: 36)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	36	-	U

COMMENTS

ROMJSB Y

FUNCTION

Converts angle in degrees to radians.

NAME	RAD1Ø
ADDRESS	53675
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Angle in degrees (8 bytes) R12 → --------------------------
OUTPUT CONDITIONS		Angle in radians (8 bytes) R12 → --------------------------

CPU CHANGES (shaded: 32–77)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Reads a string from the tape buffer and stores it in a variable area.

NAME	READ#$
ADDRESS	31335
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

Pointer to string variable area (2 bytes)
Maximum storage length (2 bytes)
Pointer to 1st character of storage (2 bytes)
R12 → ---------------------------------

OUTPUT CONDITIONS

R12 → ---------------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

A buffer must have been assigned and READ#. called before READ#$ is called.

FUNCTION

Reads a number from the tape buffer and stores it in a variable area.

NAME	READ#N
ADDRESS	31167
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

See stack requirements for STOSV

OUTPUT CONDITIONS

R12 → -------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION	NAME	REM1Ø
Returns the remainder (A,B) = A-B (IP(A/B)).	ADDRESS	51736
	TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	A-value (8 bytes) B-value (8 bytes) R12 → ----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Remainder (8 bytes) R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	12	U

COMMENTS

ROMJSB N

FUNCTION	NAME	RND1Ø
Returns a pseudo-random number between Ø and 1.	ADDRESS	53144
	TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of number	Random number (8 bytes) R12 → ----------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION	NAME	RNDIZ.
Executes the RANDOMIZE statement.	ADDRESS	55115
	TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	RANDOMIZE value (8 bytes) R12 → -------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS ROMJSB Y

RANDOMIZE value is optional.

FUNCTION	NAME	SCRAT.
Executes a SCRATCH.	ADDRESS	4437
	TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	46	36	U

COMMENTS ROMJSB Y

SCRAT. sets the immediate break bits (5 and 7) in R17.

FUNCTION

Returns secant of argument.

NAME	SEC1Ø
ADDRESS	53463
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       Argument (8 bytes)
R12 → ------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R40 = Copy of secant result

R12 STACK CONTENTS

```
       Secant result (8 bytes)
R12 → ----------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Prints a number to the display buffer or print buffer.
(Same as PRINT 5, in BASIC.)

NAME	SEMIC.
ADDRESS	70765
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       Number (8 bytes)
R12 → ----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
R12 → ----------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

DISP. or PRINT. must be called to set up select code and buffer pointers before SEMIC. is called.

FUNCTION

Prints a string to the print buffer or the display buffer.
(Same as PRINT A$; in BASIC.)

NAME	SEMIC$
ADDRESS	70643
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Length of string (2 bytes) Address of string (2 bytes) R12 → ---------------------------
OUTPUT CONDITIONS		R12 → ---------------------------

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

DISP. OR PRINT. must be called to set up select code and buffer pointers before SEMIC$ is called.

FUNCTION

SGN function; returns -1 if x<Ø, Ø if x=Ø, and +1 if x>Ø.

NAME	SGN5
ADDRESS	53405
TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-value (8 bytes) R12 → -----------------
OUTPUT CONDITIONS	R40 = Copy of SGN value	SGN value (8 bytes) R12 → -------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION

Returns the sine of the argument.

NAME SIN1Ø
ADDRESS 53546
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Argument (real or integer #) R12 → -----------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Copy of sine result	Sine result (real #) R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: D E: U
DRP: 40 ARP: 12
STATUS: U

COMMENTS

ROMJSB N

FUNCTION

Returns the square root of the argument.

NAME SQR5
ADDRESS 52442
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Argument (8 bytes) R12 → -----------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Square root (8 bytes) R12 → ---------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: D E: U
DRP: U ARP: U
STATUS: U

COMMENTS

ROMJSB N

FUNCTION

Executes standard BEEP. (BEEP with no parameters.)

NAME	STBEEP
ADDRESS	7017
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	32	33	34	35	**36**	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	31	U	U

COMMENTS

ROMJSB N

FUNCTION

Subtracts Y from X.

NAME	SUBROI
ADDRESS	52127
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
         X-value (8 bytes)
         Y-value (8 bytes)
R12 →    -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R40 = Copy of result

R12 STACK CONTENTS

```
         X-Y result (8 bytes)
R12 →    --------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	40	12	U

COMMENTS

ROMJSB N

FUNCTION	NAME	SUB1Ø
Subtracts two real numbers.	ADDRESS	52137
	TYPE	Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS

INPUT CONDITIONS	R50 = Real #A
	R40 = Real #B	

OUTPUT CONDITIONS	R40 = Real result (Copy)	Real result (A-B) R12 → -----------------

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7	D	U
10	11	12	13	14	15	16	17	DRP	ARP
20	21	22	23	24	25	26	27	U	U
30	31	32	33	34	35	36	37	STATUS	
40	41	42	43	44	45	46	47	U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS — ROMJSB: N

Not listed in global file. Same as SUBROI, except SUBROI expects real or integer numbers on the R12 stack at entry.

FUNCTION	NAME	
	ADDRESS	
	TYPE	

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7		
10	11	12	13	14	15	16	17	DRP	ARP
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	STATUS	
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS — ROMJSB:

FUNCTION	
Returns the tangent of the argument.	NAME TAN1Ø ADDRESS 53566 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Argument (real or integer #) R12 → ----------------------------
OUTPUT CONDITIONS	R40 - Copy of result	Tangent result (real #) R12 → -----------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	D	U
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	40	12
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67	STATUS	
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

FUNCTION	
Returns the current system time.	NAME TIME. ADDRESS 65517 TYPE Runtime

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS	R40 = Copy of time	Time (8 bytes) R12 → --------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	D	U
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	40	12
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67	STATUS	
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB Y

FUNCTION

Compares two strings for equality.

NAME	UNEQ$.
ADDRESS	3025
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        String 1 length (2 bytes)
        String 1 address (2 bytes)
        String 2 length (2 bytes)
        String 2 address (2 bytes)
R12 →  --------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        True/false value (8 bytes)
R12 →  --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	D	U
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	40	12
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

FUNCTION

Compares two numbers for inequality.

NAME	UNEQ.
ADDRESS	62202
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        #1 Value (8 bytes)
        #2 Value (8 bytes)
R12 →  -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        True/false value (8 bytes)
R12 →  --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	40	12
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB Y

FUNCTION

Converts all lower-case characters in a string to upper case.

NAME	UPC$.
ADDRESS	3373
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       Length of string (8 bytes)
       Address of string (8 bytes)
R12 →  ---------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       Length of string (8 bytes)
       Address of string (8 bytes)
R12 →  ---------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	**56**	**57**
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	30	U	U

COMMENTS

ROMJSB: N

FUNCTION

Converts a number into its corresponding ASCII characters.

NAME	VAL$.
ADDRESS	3207
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
       Number (8 bytes)
R12 →  ----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R26 = Address of string
R30 = Length of string

R12 STACK CONTENTS

```
       Length of string (2 bytes)
       Address of string (2 bytes)
R12 →  ---------------------------
```

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
B	U	26	12	U

COMMENTS

ROMJSB: N

FUNCTION

Converts an ASCII string of numeric characters to the corresponding numeric value.

NAME VAL.
ADDRESS 3250
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Length of string (2 bytes) Address of string (2 bytes) R12 → --------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Numeric value (8 bytes) R12 → ----------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47	U	U
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

FUNCTION

Executes the WAIT statement.

NAME WAIT.
ADDRESS 65701
TYPE Runtime

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	WAIT count (8 bytes)

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47	U	U
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

WAIT count is in milliseconds. Returns immediately if R16#2.

FUNCTION

Executes Y^X.

NAME	YTX5
ADDRESS	53242
TYPE	

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
         Y-value (8 bytes)
         X-value (8 bytes)
R12 →    -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

$$Y^X \text{ result (8 bytes)}$$

```
R12 →    ------------------
```

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB: Y

FUNCTION

Sets a specified number of bytes equal to zeros or blanks (40_8), starting at a specified address.

NAME	ZROMEM
ADDRESS	44066
TYPE	Runtime

INPUT CONDITIONS

REGISTER CONTENTS

R23 = 3 for blanks, #3 for zeros
R36 = Pointer to first byte
R56-57 = Number of bytes

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	**76**	**77**

DCM	E	DRP	ARP	STATUS
-	-	U	U	U

COMMENTS

ROMJSB: N

BIN mode should be set before entry.

GENERAL-PURPOSE UTILITY ROUTINES

The general-purpose routines on the following pages may find uses during runtime, parsing, initialization, or at other times.

FUNCTION

Compares two real numbers.

NAME	COMFLT
ADDRESS	32621
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

R40 = #A
R50 = #B	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R50 = B-A	

CPU CHANGES (shaded: 32–37, 40–47, 50–57, 60–67)

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS ROMJSB N

At output:
E = Ø if #A > = #B
E = 1 if #A < #B

FUNCTION

Converts a two-byte binary number into an eight-byte floating-point number.

NAME	CONBIN
ADDRESS	3572
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R36 = Binary #	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R40 = Floating-point #	

CPU CHANGES (shaded: 32–37, 40–47)

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS ROMJSB Y

FUNCTION

Converts real number in R60-67 to binary number in R76-77.

NAME	CONINT
ADDRESS	44321
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R60 = Real #	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R76 = Binary #	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

(Shaded: 34–37, 50–77)

DCM	E	DRP	ARP	STATUS
-	-	-	-	-

COMMENTS

ROMJSB N

Performs SAD at entry.
Performs PAD at exit.

FUNCTION

Formats a floating-point number into ASCII characters for printing.

NAME	CVNUM
ADDRESS	71135
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R30 = Pointer to output buffer. R40 = Floating-point # to be formatted.	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R30 = Pointer to next available byte in output buffer.	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

(Shaded: 0–3, 14–15, 20–77)

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Vectors output to appropriate device, obeying CRT IS and PRINT IS commands.

NAME	DRV12.
ADDRESS	5462
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R26-27 = Pointer to beginning of buffer to be output. R36-37 = Number of bytes to be output.	

OUTPUT CONDITIONS

If I/O is hooked up, assume all CPU register contents are altered; otherwise register changes shown below are correct.	

CPU CHANGES

0	1	**2**	**3**	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	**26**	**27**
30	**31**	**32**	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

Before DRV12. is called for the first time, an I/O routine such as PRINT. or DISP. should be called to initialize SCTEMP to the desired device.

DRV12. calls OUTSTR, PRDVRI, or IOTRFC.

FUNCTION

Fetches array variable value.

NAME	FETAV
ADDRESS	44727
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	See FETAVA routine.

OUTPUT CONDITIONS

R34 = Address of array variable element R60 = Value of array variable element	

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Fetches array variable address.

NAME	FETAVA
ADDRESS	44734
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        Pointer to variable area (2 bytes)
        Row dimension (8 bytes)
        Column dimension (8 bytes)
          (optional)
        Dimension flag (1 byte)
R12 →   -------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R34 = Address of array variable element

R12 STACK CONTENTS

```
R12 →   -------------------------
```

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB	N

FUNCTION

Fetches the length and absolute address of the first character of a string.

NAME	FETST
ADDRESS	45206
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        Address of name block (2 bytes)
          (Relative if program mode,
          absolute if calculator mode.)
R12 →   ------------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        Length of string (2 bytes)
        Address of string (2 bytes)
R12 →   --------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	**46**	**47**
50	51	**52**	**53**	**54**	**55**	**56**	**57**
60	61	62	63	64	65	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS

ROMJSB	N

FUNCTION

Fetches the value of a simple numeric variable. Converts integer values into tagged integers and converts short numbers to real numbers.

NAME	FETSV
ADDRESS	44535
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R66 = Address of variable Relative if program mode. Absolute if calculator mode.	
OUTPUT CONDITIONS	R34 = Absolute address of variable R46 = Name block R60 = Variable value	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	**32**	33	**34**	**35**	**36**	**37**		
40	41	42	43	44	45	**46**	**47**	U	U
50	51	52	53	54	55	56	57	STATUS	
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

FUNCTION

Returns the name block of a variable and ensures the address is absolute.

NAME	FETSVA
ADDRESS	44556
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R66 = Relative address if in program RUN mode; absolute address if in calculator mode.	
OUTPUT CONDITIONS	R46-7 = Name block of variable R34 = Absolute address of variable	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	**34**	**35**	36	37		
40	41	42	43	44	45	**46**	**47**	U	U
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

If R16 is odd, the computer is in calculator mode and the address is absolute. If R16 is even, the computer is in RUN mode and FWCURR must be added to the address. A check is also made for remote (common) variables.

FUNCTION

Performs binary integer multiplication.

NAME	INTMUL
ADDRESS	53076
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS

R66 = Multiplier
R76 = Multiplicand

R12 STACK CONTENTS

OUTPUT CONDITIONS

R54 = Result (4 bytes. Answer is full 32-bit number; the sign bit may be set.)
R66 = Multiplier
R76 = Multiplicand

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	**54**	**55**	**56**	**57**
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	-	-	-

COMMENTS

ROMJSB: N

Performs SAD at entry, PAD at exit.
Does not destroy multiplier and multiplicand.

FUNCTION

Converts a tagged BCD integer number in R60 to a real number in R60.

NAME	INTORL
ADDRESS	56343
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS

R60 = Integer #

R12 STACK CONTENTS

OUTPUT CONDITIONS

R60 = Converted real #

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	36	60	U

COMMENTS

ROMJSB: N

FUNCTION	NAME	MOVDN
	ADDRESS	37324
	TYPE	Utility

Moves a block of memory, starting with the highest address and working down to the lowest address.

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R22 = Byte count. R24 = First byte to be moved. (Highest address.) R26 = First byte of destination. (Highest address.)	

OUTPUT CONDITIONS

CPU CHANGES (shaded: 22, 23, 24, 25, 26, 27, 32, 33)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	U	U	U

COMMENTS — ROMJSB: N

Expects binary mode at entry.

FUNCTION	NAME	MOVUP
	ADDRESS	37365
	TYPE	Utility

Moves a block of memory, starting at the lowest address and working up to the highest address.

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R22 = Byte count. R24 = First byte to be moved. (Lowest address.) R26 = First byte of destination. (Lowest address.)	

OUTPUT CONDITIONS

CPU CHANGES (shaded: 22, 23, 24, 25, 26, 27, 32, 33)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	U	U	U

COMMENTS — ROMJSB: N

Expects binary mode at entry.

FUNCTION

Fetches one number from R12 stack and converts it to binary.

NAME	ONEB
ADDRESS	56113
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
        Real or integer # to pop
R12 → ------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R46-R47 = Binary # from stack
R76-R77 = Binary # from stack (Copy)

R12 STACK CONTENTS:

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	76	46	U

COMMENTS

ROMJSB: N

FUNCTION

Gets one number (off R12) as an integer.

NAME	ONEI
ADDRESS	56154
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
        Real or integer #
R12 → -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R44-R47 = Tagged BCD integer

R12 STACK CONTENTS:

```
R12 → -----------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB: N

E=Ø if valid integer.
E=1 if real number converted to integer was too large and overflowed. (In this case, R45-47 = 99999.)

FUNCTION

Fetches one real number from R12 stack.

NAME	ONER
ADDRESS	56215
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
        Real or integer # (8 bytes)
R12 →  ---------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R40 = Real #.
R60 = Real #. (Copy.)

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	Ø	60	40	U

COMMENTS — ROMJSB N

Expects DCM set to binary mode at entry.
ONER+, address 56200, has the same function, but expects real or integer number in R60-67 rather than on R12 stack. Output is the same.

FUNCTION

Gets one number (real or integer) from R12 stack and sets E flag according to type of number. Number comes off unchanged.

NAME	ONEROI
ADDRESS	56253
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
        Real or integer #
R12 →  -----------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

```
If real:    R40-47 = #
            E = Ø
If integer:   R44 = 377
              R45-47 = #
              E = 1
```

R12 STACK CONTENTS:

```
R12 →  -----------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	44	12	U

COMMENTS — ROMJSB N

E = Ø if real, 1 if integer.

FUNCTION	
Causes internal printer to advance one line.	NAME PAPER. ADDRESS 76144 TYPE Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	U	U	U

COMMENTS — ROMJSB Y

Expects binary mode at entry.

FUNCTION	
Dumps a buffer to the internal printer.	NAME PRDVR1 ADDRESS 75767 TYPE Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R26 = Address of buffer. R36 = Number of bytes in buffer.	
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	**26**	**27**
30	**31**	**32**	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS — ROMJSB Y

FUNCTION

Releases temporary scratch-pad memory.

NAME	RELMEM
ADDRESS	37534
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	**34**	**35**	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	**56**	**57**
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	U	U	U

COMMENTS

ROMJSB: N

FUNCTION

Reserves a block of memory for scratch-pad use. <u>Temporary</u> only.

NAME	RESMEM
ADDRESS	37442
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R54-55 = Number of bytes to be reserved.	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R26-27 = Address of 1st byte of reserved memory.	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	**26**	**27**
30	31	**32**	**33**	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	**56**	**57**
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	U	56	54	U

COMMENTS

ROMJSB: N

FUNCTION

ROM switching subroutine. Selects the desired ROM and executes a JSB to the desired routine in that ROM. When control is returned, reselects the calling ROM and returns.

NAME ROMJSB
ADDRESS 4776
TYPE Utility

INPUT CONDITIONS

ROMJSB calling sequence:
    JSB=ROMJSB
    Routine address (2 bytes)
    ROM# (1 byte)
ARP, DRP, and status are not preserved during the call.

During the call, ROMJSB saves RØ-1 on the R6 stack along with the ROM# of the calling ROM. (This is a total of 3 bytes plus the RTN addresses.)

OUTPUT CONDITIONS

Preserves the ARP, DRP, and status set by the called routine, and restores the original RØ.

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: U E: U
DRP: U ARP: U
STATUS: U

COMMENTS

ROMJSB -

ERTEMP (100674-100677) is destroyed.

FUNCTION

Reselects system bank-selectable ROM (ROM Ø) and returns.

NAME ROMRTN
ADDRESS 4762
TYPE Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: - E: -
DRP: Ø ARP: -
STATUS: U

COMMENTS

ROMJSB N

An external ROM would perform a GTO ROMRTN after parse routines.

FUNCTION

Reserves a block of memory for scratch-pad use. Temporary memory only.

NAME	RSMEM-
ADDRESS	37453
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R56 = Number of bytes to be reserved.	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R26-7 = Address of 1st byte of reserved memory.	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	**26**	**27**
30	31	**32**	**33**	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
–	–	–	–	–

COMMENTS

ROMJSB: N

Not in global file.
Executes a SAD at entry, a PAD at exit.

FUNCTION

NAME	
ADDRESS	
TYPE	

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB:

FUNCTION

Calculates a checksum for memory. (Especially useful for ROMs.)

NAME	RSUM#K
ADDRESS	37726
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R32 = Start address R34 = (# bytes/2) - 1	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	46	32	U

COMMENTS

ROMJSB N

Returns Z (zero flag) true if checksum is OK; otherwise Z is not true. Expects last 4 bytes of memory checked to be checksum that is compared. Expects binary mode at entry.

FUNCTION

Used by external ROMs to perform a checksum at power-on. (Checksum is for an entire 8K ROM.)

NAME	RSUM8K
ADDRESS	37722
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R32 = Start address	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	46	32	U

COMMENTS

ROMJSB N

Expects last 4 bytes of memory checked to be checksum that is compared. Expects binary mode at entry.

FUNCTION

Converts a real number to a BCD tagged integer.

NAME	RTOIN
ADDRESS	44204
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R60 = Real # to be converted	
OUTPUT CONDITIONS	R65 = BCD integer	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
D	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Scratches binary program and BASIC program. Does not reset all pointers, however. Should be used only by external ROMs that are stealing RAM at power-on.

NAME	SCRAT+
ADDRESS	4344
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	12	12	U

COMMENTS

ROMJSB N

FUNCTION

Sets bits 7 and 5 (immediate break) in R17.

NAME	SET240
ADDRESS	11243
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	**17**
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
-	-

DRP	ARP
36	6

STATUS: U

COMMENTS

ROMJSB: N

This routine is useful if it is desired that the intepreter halt when a return to it is performed.

FUNCTION

Stores a string into a string variable area; handles variable tracing if TRACE mode is active.

NAME	STOST
ADDRESS	45603
TYPE	Utility

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
          Pointer to variable area (2 bytes)
          Maximum storage length (2 bytes)
          Pointer to 1st char. of storage
            (2 bytes)
          String length (to store) (2 bytes)
          String address (to store) (2 bytes)
R12 →     ------------------------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
R12 →     ------------------------------------
```

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E
U	U

DRP	ARP
U	U

STATUS: U

COMMENTS

ROMJSB: Y

FUNCTION

Stores a value into a simple numeric or an array variable in the proper format; handles tracing if TRACE mode is active.

NAME STOSV
ADDRESS 45254
TYPE Utility

INPUT CONDITIONS

R12 STACK CONTENTS IF SIMPLE NUMERIC:
Address of variable (2 bytes)
Name block (2 bytes)
Value (8 bytes)
R12 → --------------------------------

IF ARRAY:
Address of variable (2 bytes)
Column (If tracing) (2 bytes)
Row (If tracing) (2 bytes)
Dimension flag (If tracing) (1 byte)
Name block (2 bytes)
Value (8 bytes)

OUTPUT CONDITIONS

R12 → --------------------------------

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7	-	U
10	11	12	13	14	15	16	17	DRP	ARP
20	21	22	23	24	25	26	27	-	-
30	31	32	33	34	35	36	37	STATUS	
40	41	42	43	44	45	46	47	-	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

Performs SAD at entry and PAD at exit.

Tokens 21 and 22 push all of the address and name block information onto the R12 stack, so an external routine needs to push only the value before calling STOSV.

FUNCTION

Fetches two numbers from R12 stack and converts them to binary integers.

NAME TWOB
ADDRESS 56176
TYPE Utility

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

#A (8 bytes)
#B (8 bytes)
R12 → ------------

OUTPUT CONDITIONS

R26-27 = #B in binary
R46-47 = #B in binary
R56-57 = #A in binary

R12 → ------------

CPU CHANGES

								DCM	E
0	1	2	3	4	5	6	7	B	U
10	11	12	13	14	15	16	17	DRP	ARP
20	21	22	23	24	25	26	27	46	26
30	31	32	33	34	35	36	37	STATUS	
40	41	42	43	44	45	46	47	U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Fetches two real numbers from R12 stack. If a number on stack is an integer, it is converted to a real.

NAME	TWOR
ADDRESS	56236
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		#A (8 bytes) #B (8 bytes) R12 → ------------
OUTPUT CONDITIONS	R40-47 = Real #B R50-57 = Real #A	R12 → ------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	D	Ø
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	60	40
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57	U	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Fetches two real or integer numbers off R12 stack. Converts either or both as necessary to make them both integer or both real. Status of E-register at exit indicates whether the two numbers are integer or real.

NAME	TWOROI
ADDRESS	56266
TYPE	Utility

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		#A (Real or integer.) #B (Real or integer.) R12 → --------------------
OUTPUT CONDITIONS	R40-47 = #B R50-57 = #A } Both real or both integer.	R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47	STATUS	
50	51	52	53	54	55	56	57	U	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

COMMENTS

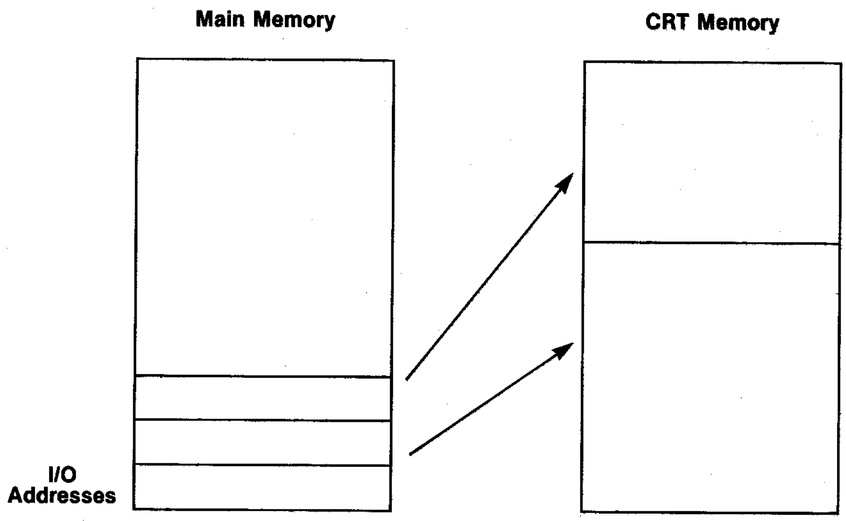
ROMJSB N

If E=Ø at exit, both numbers are real.
If E=1 at exit, both numbers are integer.

CRT CONTROL AND ROUTINES

CRT CONTROL

The memory that controls the CRT display is completely separate from the computer's main memory. This CRT memory is addressed through I/O addresses in the main memory.



There are four I/O addresses in RAM that are used to address the CRT. Each address requires a two-byte quantity to specify a CRT memory address. The I/O addresses are:

Address	Name	Function
177404	CRTSAD	Write only. Two bytes set current display start address (i.e., home address).
177405	CRTBAD	Write only. Two bytes set current byte address (i.e., cursor location). The contents of this address do not cause a cursor to appear on the CRT at that location; they merely specify the CRT location to which the next character will be output or from which the next character will be read.

177406 CRTSTS This byte defines CRT status, as shown here:

	Bit	0	1
WRITE:	0	No read request	Read request
	1	Un-wipe	Wipeout
	2	Power-up	Power-down
	3	Not used	—
	4	Not used	—
	5	Not used	—
	6	Not used	—
	7	Alpha	Graphics
READ:	0	Data not ready	Data ready to read
	1	Retrace time	Display time
	2	Not used	—
	3	Not used	—
	4	Not used	—
	5	Not used	—
	6	Not used	—
	7	Not busy	Busy

177407 CRTDAT This byte contains the data output to or read from the CRT, as shown below:

	Bit	
WRITE:	0–6	ASCII code for one byte of data
	7	= 1 causes underline (cursor)
READ:	0–6	ASCII code for one byte of data
	7	= 1 is underlined or cursor

To underline a character, the MSB of the character is set when it is output to the CRT; the character then appears on the CRT screen as if the cursor were set beneath it. A blank cursor is created by outputting a blank character with its MSB set.

Each time CRTDAT is read from or written to, the controller in CRT memory automatically increments by two the CRTBAD address. However CRTBYT (in system RAM) is not automatically updated by the CRT controller.

Because the user cannot read from I/O addresses CRTSAD or CRTBAD, and because reading from CRTSTS does not yield exactly what was written, the system normally keeps copies of the contents of these three I/O addresses elsewhere in system RAM (where they may be read). The copies are maintained in the locations shown here:

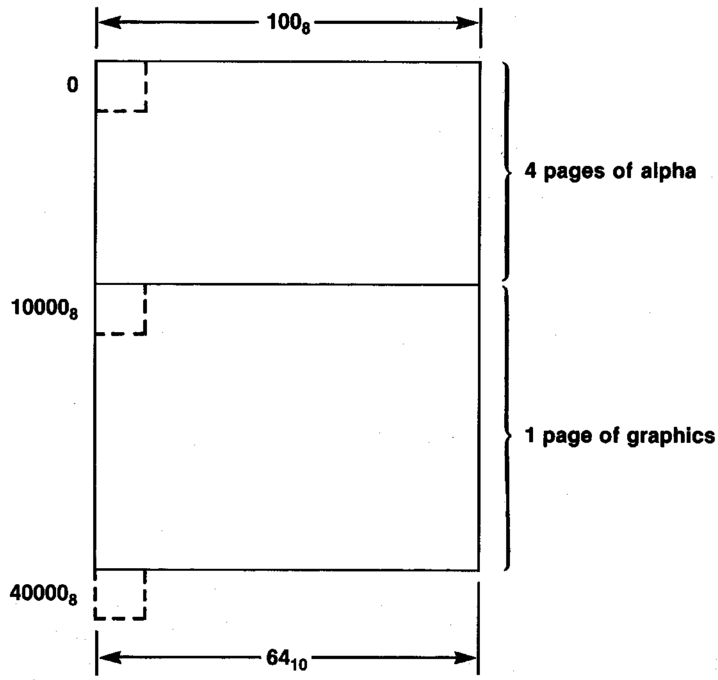
I/O Name	RAM Location Name	RAM Address
CRTSAD	CRTRAM	100200
CRTBAD	CRTBYT	100176
CRTSTS	CRTWRS	101016

CRT ADDRESSING

The CRT memory employs "nibble addressing"--each address in the CRT memory contains only four bits. Such an addressing scheme provides greater resolution and control over the CRT display.

When sending information to CRT memory, the system must output the contents of a complete eight-bit byte. Thus, each byte shipped is stored at two consecutive addresses in CRT memory. The most significant four bits are stored at the lower-numbered CRT memory address, and the least significant four bits are stored at the higher-numbered address in CRT memory.

CRT memory is partitioned into alpha and graphics areas.



Alpha Display: Alpha addresses in CRT memory are from 0 to 7777_8. In alpha mode the display shows 16_{10} lines of 32_{10} characters per line. The scrolling keys permit the user to view an additional 48_{10} lines of alphanumeric data. Thus, only 1/4 of the information in the alpha area of CRT memory fits on the CRT screen at any one time.

Each ASCII character occupies eight bits. Because of its nibble addressing, two address locations in CRT memory are required to store one ASCII character.

In alpha mode, one character occupies a space on the CRT of 8_{10} dots by 12_{10} dots. In alpha mode, the screen can contain 16_{10} rows, with 32 (i.e., 40_8) characters per row.

For example, the following section of code will output a character to the 2nd row down, 4th character in the row, of the CRT screen:

```
LDM 34, = 106, Ø          Loads desired CRT memory address.
JSB = BYTCRT              Sets CRTBYT and CRTBAD to specified address.
LDB R32, = 101            Loads character (A) to ship out.
JSB = CHKSTS          }
STBD 32, = CRTDAT     }   When CRT controller not busy, byte is output.
```

An alternate method of executing the last two instructions (JSB = CHKSTS and STBD 32, = CRTDAT) would be JSB = OUTCHR. This method may be preferable, since OUTCHR automatically updates CRTBYT and CRTBAD to the next consecutive location.

Graphics Display: Graphics addresses in CRT memory are from 10000 to 37777_8. The graphics display mode, which is entered when the user presses the [GRAPH] key or executes a graphics statement, shows all information in the graphics area of CRT memory at one time. In graphics mode, the screen has a resolution of 256_{10} dots wide by 192_{10} dots high. Any consecutive pair of four-bit nibble addresses in CRT memory can be specified. The address of the first nibble is specified by CRTBAD. Thus, each byte of information output from the CPU to CRT memory controls eight dots (i.e., two four-bit nibbles) on the CRT.

For example, the following section of code outputs one byte to addresses 10224 and 10225 of CRT graphics memory.

```
LDM R34, = 224, 20    }
JSB = BYTCRT          }   Set CRTBAD and CRTBYT to 10224.
LDB R32, = 27             Byte to be output.
JSB = CHKSTS
STBD R32, = CRTDAT
```

In the CRT, the byte shipped out affects address 10224. This is the third row from the top of the graphics CRT, the 80th through the 87th dots from the left.

CRT ROUTINES

System routines useful in CRT control follow.

FUNCTION	
Forces CRT to alpha mode, if alpha mode is not already active.	NAME ALPHA. ADDRESS 36105 TYPE CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	31	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS ROMJSB N

CRT is in alpha mode at exit.

FUNCTION

Extends blanks (carriage returns) to remainder of line on CRT. Does not update CRTBYT, but cursor is at start of next line insofar as CRT controller is concerned.

NAME	BLKLIN
ADDRESS	36320
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS	R34 = Current cursor location (CRTBYT)	
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	**31**	**32**	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	U	U	U

COMMENTS

Use CLREOL if updating of CRTBYT is desired. Z is true at exit.

ROMJSB N

FUNCTION

Executes the BPLOT statement.

NAME	BPLOT.
ADDRESS	34365
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		Length of string (2 bytes) Address of string (2 bytes) # Bytes/line (8 bytes) R12 → --------------------------
OUTPUT CONDITIONS		R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	32	33	**34**	**35**	**36**	**37**
40	41	42	43	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Same as BPLOT statement, but with parameters in registers rather than on stack.

NAME	BPLOT+
ADDRESS	34405
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R22-3 = Length of string
R34-5 = Address of string
R44-5 = # Bytes/line
R46-7 = # Bytes/line (copy)

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS

ROMJSB: N

Not in global file.
Does <u>not</u> switch to graphics mode if not already there.

FUNCTION

NAME	
ADDRESS	
TYPE	

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB

FUNCTION

Moves cursor to position specified by the register pair specified by the DRP setting at entry.

NAME	BYTCRT
ADDRESS	35423
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

DRP register pair = Address to which to move cursor

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	-	-
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	-	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

DRP at exit is the same as at entry.

FUNCTION

Moves cursor to the specified position, but does not generate cursor on CRT screen.

NAME	BYTCR!
ADDRESS	35422
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R34-35 = Address to which cursor is to be moved

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	24	25	26	27		
30	31	32	33	34	35	36	37	DRP	ARP
40	41	42	43	44	45	46	47	34	-
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Loops until CRT is not busy.

NAME CHKSTS
ADDRESS 36335
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R30 = CRTSTS	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: B E: -
DRP: 30 ARP: -
STATUS: U

COMMENTS

ROMJSB N

At exit, CRT is ready to accept an address or a byte of data.

FUNCTION

Forces CRT screen to alpha mode, clears screen to blanks (carriage returns), and homes the cursor.

NAME CLEAR.
ADDRESS 35021
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM: B E: -
DRP: U ARP: U
STATUS: U

COMMENTS

ROMJSB N

FUNCTION

Extends blanks (carriage returns) to end of line, but leaves cursor at its current position at entry.

NAME	CLREOL
ADDRESS	35535
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R32 = 15 R66-67 = CRTBYT	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	U
20	21	22	23	**24**	**25**	26	27	DRP	ARP
30	**31**	**32**	33	**34**	**35**	36	37	34	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	**66**	**67**		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

FUNCTION

Counts CRT retraces (the number in R31 at entry) and returns.

NAME	CNTRTR
ADDRESS	36002
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R31 = Number of retraces to count	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R31 = Ø	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	24	25	26	27	DRP	ARP
30	**31**	32	33	34	35	36	37	31	-
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77	U	

COMMENTS

ROMJSB N

Z = 1 (true) at exit.
There are 60 retraces per second.

FUNCTION

Executes COPY command.

NAME COPY.
ADDRESS 75360
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS | R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	**26**	**27**
30	**31**	**32**	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	**56**	**57**
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Initializes portion of CRT alpha to blanks (carriage returns).

NAME CRTBL+
ADDRESS 36255
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS

R34-35 = Starting address (1st byte to blank)
R36-37 = Number of bytes to blank

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	33	**34**	**35**	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	36	U	U

COMMENTS

ROMJSB N

FUNCTION	NAME CRTBLK ADDRESS 36247 TYPE CRT
Blanks all four pages of alpha screen with carriage returns, and homes cursor.	

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	33	**34**	**35**	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	36	U	U

COMMENTS

ROMJSB N

FUNCTION	NAME CRTINT ADDRESS 36177 TYPE CRT
Initializes CRT: clears all of alpha and graphics, and homes cursor in alpha mode.	

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	**31**	**32**	33	**34**	**35**	**36**	**37**
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
B	-	31	U	U

COMMENTS

ROMJSB N

FUNCTION

Powers down the CRT. (Must be performed before driving the printer or tape.)

NAME	CRTPOF
ADDRESS	35703
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

(30, 31 shaded)

DCM	E	DRP	ARP	STATUS
B	-	30	31	U

COMMENTS

ROMJSB: N

LSB is even at exit.

FUNCTION

Powers up CRT. (Also powers down tape transport and waits for printer to be not busy.)

NAME	CRTPUP
ADDRESS	35716
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

(30, 31 shaded)

DCM	E	DRP	ARP	STATUS
B	-	31	-	U

COMMENTS

ROMJSB: Y

FUNCTION

Un-wipes CRT. (See CRTWPO.)

NAME	CRTUNW
ADDRESS	36067
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	31	-	U

COMMENTS

ROMJSB: N

FUNCTION

Wipes out CRT display. Does not cause power-down, and no data is lost from screen. (Often used to eliminate screen flashes and to speed up transfer of data.)

NAME	CRTWPO
ADDRESS	35661
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	30	31	U

COMMENTS

LSB is even at exit.

ROMJSB: N

FUNCTION	NAME	CURS
Generates a cursor at current CRTBYT address.	ADDRESS	35055
	TYPE	CRT

		R12 STACK CONTENTS
INPUT CONDITIONS	CRTBYT (RAM location) = Current cursor location	
OUTPUT CONDITIONS	Cursor generated on CRT at CRTBYT address	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	**24**	**25**	26	27	DRP	ARP
30	**31**	**32**	**33**	**34**	**35**	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION	NAME	DECUR2
Removes two cursors from the CRT.	ADDRESS	35547
	TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	-
20	21	22	23	**24**	**25**	26	27	DRP	ARP
30	31	**32**	**33**	**34**	**35**	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	**66**	**67**	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION

Moves cursor down one line. If cursor would move off bottom, it wraps around to the top line of the current screen.

NAME	DNCUR.
ADDRESS	35306
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES (shaded: 24, 25, 30, 34, 35)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
34	24
STATUS	
U	

COMMENTS ROMJSB N

Does not generate cursor on screen.

FUNCTION

Moves cursor down one position. Does not wrap around on current page, but does wrap from bottom of alpha to top of alpha.

NAME	DNCURS
ADDRESS	35370
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES (shaded: 24, 25, 30, 34, 35)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
34	24
STATUS	
U	

COMMENTS ROMJSB N

Does not generate cursor on screen.

FUNCTION	NAME DRAW.
Draws a line from the current pen position to the specified point. (For CRT only.)	ADDRESS 33015 TYPE CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		X-coordinate (8 bytes) Y-coordinate (8 bytes) R12 → ---------------------
OUTPUT CONDITIONS		R12 → ---------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

FUNCTION	NAME EOJ2
Clears keyboard interrupt bit in SVCWRD, and clears break bit in R17 if no other interrupts are pending. Also sets key repeat counter.	ADDRESS 34772 TYPE CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS	R32 = KRPET1 R33 = SVCWRD	

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	-	-
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	32	32
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB N

If an external routine takes over CHIDLE to handle a key itself, the routine must call EOJ2 before popping off two returns and returning; otherwise, it appears to the system as though the key has not been handled yet, and the system will keep looping back.

FUNCTION

Performs a keyboard FLIP.

NAME	FLIP.
ADDRESS	35011
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R36 = 200	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
-	-	36	-	U

COMMENTS

ROMJSB N

FUNCTION

Forces graphics mode and clears graphics screen. Can have one optional parameter on R12 stack.

NAME	GCLR.
ADDRESS	36013
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	31	U	U

COMMENTS

ROMJSB N

WARNING

Checks R12 against TOS to determine if there is an optional parameter on the stack. So if something else (not intended for GCLR.) is on the R12 stack, save TOS on the R6 stack and set TOS = R12 before calling GCLR. Then recover TOS from the R6 stack.

FUNCTION

Forces CRT to graphics display mode.

NAME	GRAPH.
ADDRESS	36147
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R34 = Ø R35 = 20	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	31	31	U

COMMENTS

ROMJSB N

FUNCTION

Clears graphics screen to appropriate pen condition. (Will cause flash if CRT is not wiped out.)

NAME	GRINIT
ADDRESS	36220
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	U	U	U

COMMENTS

Expects binary mode at entry.

ROMJSB N

FUNCTION

Outputs a string to the CRT without performing a carriage return. (Does not fill with blanks to the end of the line.)

NAME	HLFLIN
ADDRESS	35121
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R26-27 = Pointer to 1st character of buffer R36-37 = Number of bytes in buffer	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R24 = 2 R25 = Ø R30 = CRTSTS R32 = Last byte output R34-35 = CRTBYT (New cursor location) R36 = Ø R37 = Ø	

CPU CHANGES (shaded: 24, 25, 26, 27, 30, 31, 32, 34, 35, 36, 37)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Moves cursor to home position on current CRT page, but does not generate cursor on CRT.

NAME	HMCURS
ADDRESS	35527
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES (shaded: 30, 34, 35)

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	34	-	U

COMMENTS

ROMJSB N

FUNCTION	NAME	IDRAW.
Performs an incremental draw from the current pen position. (CRT only.)	ADDRESS	32752
	TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-increment (8 bytes) Y-increment (8 bytes) R12 → --------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION	NAME	IMOVE.
Executes the IMOVE statement.	ADDRESS	31675
	TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-coordinate (8 bytes) Y-coordinate (8 bytes) R12 → --------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Inputs one character from current byte address of CRT.

NAME	INCHR
ADDRESS	35244
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS	R32 = Character from CRT	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
32	-
STATUS	
U	

COMMENTS

ROMJSB N

FUNCTION

Same function as INCHR if CRT is wiped out. (INCHR- saves time, but should not be used unless it is guaranteed that the CRT is wiped out.)

NAME	INCHR-
ADDRESS	35220
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
32	-
STATUS	
U	

COMMENTS

ROMJSB N

FUNCTION

Executes a LABEL statement to the CRT.

NAME	LABEL.
ADDRESS	34044
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Length of string (2 bytes) Address of string (2 bytes) R12 → ---------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ---------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB: Y

FUNCTION

Sets label direction for CRT graphics.

NAME	LDIR.
ADDRESS	34020
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	LDIR angle (8 bytes) R12 → --------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	45	47	U

COMMENTS

ROMJSB: N

FUNCTION

Moves cursor left one position on current display. If cursor moves off left end of top line, it wraps around to right of bottom line.

NAME	LTCUR.
ADDRESS	35332
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
34	24
STATUS	
U	

COMMENTS — ROMJSB: N

Does not generate cursor on CRT screen.

FUNCTION

Moves cursor left one position. Cursor does not wrap around on current page.

NAME	LTCURS
ADDRESS	35376
TYPE	CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	**ARP**
34	24
STATUS	
U	

COMMENTS — ROMJSB: N

FUNCTION	NAME	MOVCRS
Moves cursor an incremental number of positions from its current position. Cursor does not wrap around on current page, but does remain on alpha screen.	ADDRESS	35410
	TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R24-25 = Twice (2*) the number of positions to move. (Two's complement for a negative value.)	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7	DCM: B	E: -
10	11	12	13	14	15	16	17		
20	21	22	23	24	25	26	27	DRP: 34	ARP: 24
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47	STATUS: U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

(Shaded: 30, 34)

COMMENTS

ROMJSB N

FUNCTION	NAME	MOVE.
Executes the MOVE statement.	ADDRESS	31703
	TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	X-coordinate (8 bytes) Y-coordinate (8 bytes) R12 → ----------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ----------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM: U	E: U
10	11	12	13	14	15	16	17		
20	21	22	23	24	25	26	27	DRP: U	ARP: U
30	31	32	33	34	35	36	37		
40	41	42	43	44	45	46	47	STATUS: U	
50	51	52	53	54	55	56	57		
60	61	62	63	64	65	66	67		
70	71	72	73	74	75	76	77		

(Shaded: 0–3, 14, 15, 20–77)

COMMENTS

ROMJSB N

FUNCTION

Outputs a single character to the CRT at the current alpha cursor position, then advances the cursor position.

NAME	OUTCHR
ADDRESS	35114
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R32 = Byte to be output	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R24-25 = 2 R30 = CRTSTS R32 = Byte that was output R34-35 = CRTBYT (New cursor location)	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	**31**	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	34	24	U

COMMENTS

ROMJSB N

FUNCTION

Outputs a buffer to the CRT and executes a carriage return. Also blank fills to the end of the output line.

NAME	OUTSTR
ADDRESS	35052
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R26-27 = Pointer to 1st character R36-37 = Number (in binary) of characters to be output.	

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	**26**	**27**
30	**31**	**32**	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	U	U	U

COMMENTS

ROMJSB N

Sets binary mode before exit.

FUNCTION

Selects graphics pen. (CRT only.)

NAME	PEN.
ADDRESS	66416
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      Pen # (8 bytes)
R12 → ---------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
R12 → ---------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	51	52	53	54	55	56	57
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E
B	U

DRP	ARP
47	40

STATUS: U

COMMENTS

ROMJSB: Y

FUNCTION

Executes the PENUP statement. (For CRT only.)

NAME	PENUP.
ADDRESS	66440
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
-	-

DRP	ARP
30	-

STATUS: U

COMMENTS

ROMJSB: Y

FUNCTION

Executes the PLOT statement.

NAME	PLOT.
ADDRESS	32642
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
        X-coordinate (8 bytes)
        Y-coordinate (8 bytes)
R12 →   ----------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
R12 →   ----------------------
```

CPU CHANGES

0	**1**	**2**	**3**	4	5	6	7
10	11	12	13	**14**	**15**	16	17
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB N

FUNCTION

Moves cursor right one position on current CRT page. From extreme bottom right, cursor wraps around to top left.

NAME	RTCUR.
ADDRESS	35351
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	34	24	U

COMMENTS

Does not generate cursor on CRT screen.

ROMJSB N

FUNCTION

Moves cursor right one position. Cursor does not wrap around on current CRT page.

NAME RTCURS
ADDRESS 35404
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-
DRP	ARP
34	24
STATUS	
U	

COMMENTS

ROMJSB N

Does not generate cursor on CRT screen.

FUNCTION

Executes the SCALE statement. (For CRT only.)

NAME SCALE.
ADDRESS 66247
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

X-minimum (8 bytes)
X-maximum (8 bytes)
Y-minimum (8 bytes)
Y-maximum (8 bytes)
R12 → ------------------

OUTPUT CONDITIONS

R12 → ------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	**21**	**22**	**23**	24	25	26	27
30	31	**32**	**33**	**34**	**35**	**36**	**37**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**
70	71	72	73	74	75	76	77

DCM	E
U	U
DRP	ARP
U	U
STATUS	
U	

COMMENTS

ROMJSB Y

FUNCTION

Scrolls CRT down one line, leaving cursor in same relative position on CRT.

NAME SCRDN
ADDRESS 35625
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R34-35 = CRTRAM	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-

DRP	ARP
34	24

STATUS: U

COMMENTS

ROMJSB N

FUNCTION

Scrolls CRT up one line, leaving cursor in same relative position on CRT.

NAME SCRUP
ADDRESS 35654
TYPE CRT

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
R34-35 = CRTRAM	

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E
B	-

DRP	ARP
34	24

STATUS: U

COMMENTS

ROMJSB N

FUNCTION

Moves cursor up one line on current page. From top line of page, cursor wraps around to bottom line.

NAME UPCUR.
ADDRESS 35264
TYPE CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	34	24	U

COMMENTS

ROMJSB N

Does not generate cursor on screen.

FUNCTION

Moves cursor up one position. Cursor does not wrap around on current page, but does wrap around from top of alpha to bottom of alpha.

NAME UPCURS
ADDRESS 35362
TYPE CRT

	REGISTER CONTENTS	R12 STACK CONTENTS
INPUT CONDITIONS		
OUTPUT CONDITIONS		

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	**24**	**25**	26	27
30	31	32	33	**34**	**35**	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	-	34	24	U

COMMENTS

ROMJSB N

Does not generate cursor on CRT screen.

FUNCTION

Executes the XAXIS statement. (For CRT only.)

NAME	XAXIS.
ADDRESS	32303
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
     Y-intercept (8 bytes)
     Tic spacing (8 bytes)
     X-minimum (8 bytes)
     X-maximum (8 bytes)
R12 → --------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
R12 → --------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS ROMJSB N

Only the Y-intercept is required. The other three parameters are optional.

FUNCTION

Executes the YAXIS statement. (For CRT only.)

NAME	YAXIS.
ADDRESS	32347
TYPE	CRT

INPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
     X-intercept (8 bytes)
     Tic spacing (8 bytes)
     Y-minimum (8 bytes)
     Y-maximum (8 bytes)
R12 → --------------------
```

OUTPUT CONDITIONS

REGISTER CONTENTS:

R12 STACK CONTENTS:

```
R12 → --------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS ROMJSB N

Only X-intercept is required. The other three parameters are optional.

TAPE CONTROL ROUTINES

Routines which provide the major entry points for control of a tape cartridge follow. In general, each of these routines expects an argument to be on the R12 stack when the routine is called.

FUNCTION	NAME	ASIGN.
Assigns a buffer to a data file.	ADDRESS	27056
	TYPE	Tape

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Buffer # (8 bytes) File name length (2 bytes) File name address (2 bytes) R12 → --------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION	NAME	CREAT.
Creates a data file.	ADDRESS	26561
	TYPE	Tape

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	File name length (2 bytes) File name address (2 bytes) # Records (8 bytes) # Bytes/record (8 bytes) R12 → --------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → --------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Prints an entire array to a tape data file.

NAME P#ARAY
ADDRESS 57642
TYPE Tape

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

Address of array (2 bytes)
Name block (2 bytes)
R12 → ---------------------------

OUTPUT CONDITIONS

R12 → ---------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	B	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

FUNCTION

Move the print pointers in the buffer. Executes the PRINT#1, or PRINT#1,1, portion of a serial or random PRINT to a data file on tape cartridge.

NAME PRNT#.
ADDRESS 30055
TYPE Tape

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

Assign buffer # (8 bytes)
Record # if random (8 bytes)
(optional)
R12 → ---------------------------

OUTPUT CONDITIONS

R12 → ---------------------------

CPU CHANGES

0	1	2	3	4	5	6	7	DCM	E
10	11	12	13	14	15	16	17	U	U
20	21	22	23	24	25	26	27	DRP	ARP
30	31	32	33	34	35	36	37	U	U
40	41	42	43	44	45	46	47		
50	51	52	53	54	55	56	57	STATUS	
60	61	62	63	64	65	66	67	U	
70	71	72	73	74	75	76	77		

COMMENTS

ROMJSB Y

FUNCTION

Purge a file.

NAME	PURGE.
ADDRESS	26013
TYPE	Tape

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      File name length (2 bytes)
      File name address (2 bytes)
      ALL flag (8 bytes)
R12 → --------------------------
      (ALL flag is optional.  See PURGE
      command in computer owner's
      handbook.)
```

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Reads an entire array from a tape data file.

NAME	R#ARAY
ADDRESS	77602
TYPE	Tape

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

```
      Address of array (2 bytes)
      Name block (2 bytes)
R12 → -------------------------
```

OUTPUT CONDITIONS

```
R12 → -------------------------
```

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
B	U	U	U	U

COMMENTS

ROMJSB Y

FUNCTION

Executes the READ#1, or READ#1,1, portion of a serial or random READ from a data file on a tape cartridge.

NAME READ#.
ADDRESS 30055
TYPE Tape

INPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	Assign buffer # (8 bytes) Record # (8 bytes) (optional) R12 → ------------------------------

OUTPUT CONDITIONS

REGISTER CONTENTS	R12 STACK CONTENTS
	R12 → ------------------------------

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS
U	U	U	U	U

COMMENTS

See READN. and READ$.

ROMJSB Y

FUNCTION

NAME
ADDRESS
TYPE

INPUT CONDITIONS

REGISTER CONTENTS

R12 STACK CONTENTS

OUTPUT CONDITIONS

CPU CHANGES

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

DCM	E	DRP	ARP	STATUS

COMMENTS

ROMJSB

DECOMPILING

Decompiling is a two-stack operation. The increasing stack pointed to by CPU register R12 is used as the expression stack, while the increasing stack addressed by R30 is used as the output buffer.

Decompiling uses system routines, especially utility routines such as for fetching variable addresses, that will be found in other areas of this section.

SECTION 8

SAMPLE BINARY PROGRAMS

This section is made up of six prewritten binary programs. In addition to being listed here, these programs are available on the tape cartridge and disc that accompany the Assembler ROM. On the cartridge and disc, source code file names end in "S," while those of object code files end in "B."

Each of these programs is designed to illustrate a facet of assembly-language programming on the HP-83/85, and each provides a function or keyword that is itself useful to the HP-83/85 operating system.

Each program listing contains at the end a table of addresses of HP-83/85 system routines that are used by the program. Inserting the Assembler Global File tape cartridge or disc and placing a GLO GLOBAL pseudo-op near the beginning of the program obviates the need for this list of addresses in some of the example programs. (Certain example programs, however, call system routines whose addresses are not available on the Assembler Global File tape cartridge or disc.)

FAHRENHEIT TO CELSIUS

FTOC BINARY
Source File: FTOCS
Object File: FTOCB

This program provides a single system function, FTOC, that converts values of temperatures in degrees Fahrenheit to degrees Celsius. Its source code and object code may be found listed in section 6 of this manual.

SOFT KEYS AS TYPING AIDS

SOFTKEY BINARY
Source File: SOFTKS
Object File: SOFTKB

This program permits each special function, or "soft," key ([K1], [K2], etc.) to contain a string of up to 95 characters; the characters are all output when the key is pressed.

The program implements a single BASIC statement:

Format: SOFTKEY key #, endline code, "text"

Description: Key # is a one-digit code (1-8) that selects the special function key. Endline code can be either Ø, to indicate text is followed by an [END LINE]; or 1, to indicate text is not followed by an [END LINE]. Text can be a string of up to 95 characters.

If text is followed by an [END LINE] (i.e., an endline code of Ø is specified) the text must be an expression, BASIC statement, program line, etc., that can be understood and parsed by the HP-83/85. The expression, statement, etc. will be executed immediately when the specified special function key is pressed.

The program takes over the character idle hook CHIDLE, and it also contains its own error messages.

```
1 !****************************
2 !*     SOFTKEY BINARY        *
3 !*  (c) Hewlett-Packard Co.  *
4 !*          1980             *
5 !****************************
10 ! FORMAT OF COMMAND IS:
20 ! SOFTKEY <NUMEXP>,<NUMEXP>,
30 !                <STREXP>
40 ! THE FIRST NUMEXP SELECTS
50 ! THE KEY, AND THE SECOND
60 ! SELECTS WHETHER THE TEXT IS
70 ! FOLLOWED BY AN ENDLINE (0)
80 ! OR NOT (NOT=1).  THE STRING
90 ! IS THE TEXT ON THE KEY.
100         NAM SOFTKY
110         DEF RUNTIM
120         DEF ASCIIS
130         DEF PARSE
140         DEF ERMSG
150         DEF INIT
160 PARSE   BYT 0,0
170         DEF SOFTK,
180 RUNTIM BYT 0,0
190         DEF SOFTK.
200         BYT 377,377
210 SOFTK, PUBD R43,+R6
220         JSB =NUMVA+
230         JSB =GETCMA
240         JSB =NUMVAL
250         JSB =GETCMA
260         JSB =STREXP
270         POBD R47,-R6
280         LDB R45,=371
290         PUMD R45,+R12
300         RTN
310 ASCIIS ASP "SOFTKEY"
320         BYT 377
330 ERMSG   BYT 200,200,200,200,200,200,200,200,200
340         ASP "SOFTKEY NUMBER OUT OF RANGE "
350         ASP "SOFTKEY STRING TOO LONG "
360         BYT 377
370 INIT    LDBD R0,=ROMFL
380         CMB R0,=1
390         JZR INITAL
400         CMB R0,=5
410         JZR RTNRTN
420         CMB R0,=2
430         JZR RTNRTN
440         CMB R0,=3
450         JZR INITAL
460         RTN
470 INITAL LDM R34,=KEYHAT
480         ADMD R34,=BINTAB
490         LDB R74,=316
500         STM R34,R75
510         LDB R77,=236
520         STMD R74,=CHIDLE
530         RTN
540 RTNRTN LDB R34,=236
550         STBD R34,=CHIDLE
```

```
560          RTN
570 ! GET BINARY KEY# OFF STACK
580 !CHECK FOR CORRECT RANGE,    RETURN ABSOLUTE ADDRESS OF KEY  STORAGE IN R46,R
47
590 KEY#     JSB =ONEB
600          CMM R46,=1,0          !IS KEY#<1?
610          JNC ERROR1            !JIF YES
620          CMM R46,=11,0         !IS KEY#>=9?
630          JCY ERROR1            !JIF YES
640          LLM R46               !DOUBLE FOR TABLE
650          ADMD R46,=BINTAB      !MAKE KEY# ABSOLUTE
660          LDMD R46,X46,KEYTBL   !LOAD ADDRESS OF KEY STORAGE
670          ADMD R46,=BINTAB      !MAKE IT ABSOLUTE
680          RTN
690 ERROR1 JSB =ERROR+             !KEY NUMBER OUT OF RANGE
700          BYT 366
710 ERROR2 JSB =ERROR+             !STRING TOO LONG
720          BYT 365
730          BYT 241               !BASIC COMMAND ATTRIBUTE
740 SOFTK. BIN
750          POMD R32,-R12         !GET STRING ADDRESS
760          POMD R30,-R12         !GET STRING LENGTH
770          CMM R30,=1,0          !IS IT A NULL OR 0-LENGTH STRING?
780          JCY CHNGKY            !JIF NO
790          POMD R40,-R12         !TRASH IMMEDIATE EXECUTION PARAMETER
800          LDM R36,=KEY#         !GET ADDRESS OF SUBROUTINE
810          ADMD R36,=BINTAB      !MAKE IT ABSOLUTE
820          JSB X36,ZERO          !JUMP TO IT
830          CLB R45               !CLEAR LENGTH OF KEY STRING=>NOTHING THERE
840          STBD R45,R46
850          RTN                   !DONE
860 CHNGKY CMM R30,=140,0          !IS LENGTH>=96?
870          JCY ERROR2            !JIF YES
880          JSB =ONEB             !GET IMMEDIATE KEY-EXECUTION VALUE FROM STACK
890          LDM R26,R46           !SAVE IN R26
900          LDM R36,=KEY#         !GET ADDRESS OF SUBROUTINE
910          ADMD R36,=BINTAB      !MAKE IT ABSOLUTE
920          JSB X36,ZERO          !JUMP TO IT
930          CMM R26,=0,0          !SHOULD IT BE IMMED. EXEC.?
940          JNZ AROUND            !JIF NO
950          LDB R31,=200          !SET PARITY BIT
960          ORB R30,R31           !SET PARITY BIT=>IMMED. EXEC.
970 AROUND PUBD R30,+R46           !SAVE LENGTH AND POINT TO START OF STRING STORAG
E
980          ANM R30,=177,0        !CLEAR OFF IMMED. EXEC. BIT, IF ANY
990 LOOP     POBD R26,+R32         !GET BYTE OF STRING
1000          PUBD R26,+R46        !SAVE IT
1010          DCB R30              !DONE YET?
1020          JNZ LOOP             !JIF NO
1030 KEYRTN RTN                    !DONE
1040 KEYHAT BIN
1050          CLM R26
1060          LDBD R26,=KEYHIT     !LOAD KEY CODE
1070          CMB R26,=200         !IS IT < 200?
1080          JNC KEYRTN           !JIF YES
1090          CMB R26,=210         !IS IT >=210?
1100          JCY KEYRTN           !JIF YES
1110          SBM R26,=177,0       !GET TO KEY#
1120          LLM R26              !DOUBLE # FOR TABLE
1130          ADMD R26,=BINTAB     !MAKE IT ABSOLUTE
1140          LDMD R26,X26,KEYTBL  !GET ADDRESS OF KEY STORAGE
1150          ADMD R26,=BINTAB     !MAKE IT ABSOLUTE
1160          POBD R36,+R26        !GET LENGTH
```

```
                CMB R36,=0          !IS IT EMPTY?
                JNZ NEXT            !JIF NO
                RTN                 !LET SYSTEM HANDLE IT
        NEXT    STB R36,R77         !SAVE FOR LATER
                ANM R36,=177,0      !MASK OFF IMMED. EXEC. BIT
                JSB =HLFLIN         !OUTPUT KEY STRING
                JSB =CURS           !SPIT OUT CURSOR
                TSB R77             !IS IMMED. EXEC BIT SET?
                JNG OUTCR           !JIF YES
                CLE                 !DONE WITH KEY
                JSB =EOJ2           !CLEAN UP
                POMD R74,-R6        !TRASH 2 RETURNS
                RTN
        OUTCR   LDB R26,=232        !LOAD ENDLINE
                STBD R26,=KEYHIT    !PUT ENDLINE IN KEYHIT
                RTN                 !LET SYSTEM HANDLE IT
        KEYTBL  BYT 0,0
                DEF K1
                DEF K2
                DEF K3
                DEF K4
                DEF K5
                DEF K6
                DEF K7
                DEF K8
        K1      BYT 2
                ASC "K1"
                BSZ 140
        K2      BYT 2
                ASC "K2"
                BSZ 140
        K3      BYT 2
                ASC "K3"
                BSZ 140
        K4      BYT 2
                ASC "K4"
                BSZ 140
        K5      BYT 2
                ASC "K5"
                BSZ 140
        K6      BYT 2
                ASC "K6"
                BSZ 140
        K7      BYT 2
                ASC "K7"
                BSZ 140
        K8      BYT 2
                ASC "K8"
                BSZ 140
        CURS    DAD 35055
        CHIDLE  DAD 102416
        KEYHIT  DAD 100671
        BINTAB  DAD 101233
        HLFLIN  DAD 35121
        EOJ2    DAD 34772
        ROMFL   DAD 101231
        OUTCHR  DAD 35114
        NUMVA+  DAD 12407
        GETCMA  DAD 13414
        NUMVAL  DAD 12412
        STREXP  DAD 13626
```

```
1780 ONEB    DAD 56113
1790 ERROR+ DAD 6611
1800 ZERO    EQU 0
1810         FIN
```

STRING UNDERLINE

STRING UNDERLINE BINARY PROGRAM
Source File: UDL$S
Object File: UDL$B

When passed one string parameter, this program returns the same string with all characters underlined. It implements a BASIC string function with one string parameter.

Format: UDL$ ("string expression")

Description: Returns the same string expression with all characters underlined.

```
10 !****************************
20 !*     STRING  UNDERLINE    *
30 !* (c) Hewlett-Packard Co. *
40 !*          1980           *
50 !****************************
60          NAM UDLBIN               !SET UP PROGRAM CONTROL BLOCK
70          DEF RUNTIM               !PTR TO RUNTIME ADDRESS TABLE
80          DEF ASCIIS               !PTR TO KEYWORD TABLE
90          DEF PARSE                !PTR TO PARSE ADDRESS TABLE
100         DEF ERMSG                !PTR TO ERROR MESSAGE TABLE
110         DEF INIT                 !PTR TO INIT ROUTINE FOR SYSTEM
120 !****************************************************************************
*
130 PARSE   BYT 0,0                  !DUMMY TOK #0 PARSE PTR
140 RUNTIM BYT 0,0                   !DUMMY TOK #0 RINTIME
150         DEF UDL$.                !TOK #1 RUNTIME
160         BYT 377,377              !TERMINATE RELOCATABLES
170 !****************************************************************************

180 ASCIIS ASP "UDL$"                !KEYWORD #1
190         BYT 377                  !TERMINATE ASCIIS TABLE
200 !****************************************************************************

210 ERMSG   BSZ 0
220         BYT 377                  !NO ERROR MESSAGES
230 !****************************************************************************

240 INIT    BSZ 0                    !NO INITIALIZATION TO BE DONE
250         RTN                      !DONE
260 !****************************************************************************

270         BYT 30,56                ! ATTRIBUTES (STRING FUNCTION, 1 STRING PARAMETE
R)
280 UDL$.   POMD R36,-R12            !POP STRING ADDRESS OFF OF R12 STACK
290         POMD R30,-R12            !POP STRING LENGTH OFF OF R12 STACK
300         STM R30,R56              !LENGTH NEEDS TO BE IN 56 FOR 'RSMEM-'
310         JSB =RSMEM-              !GO GET SOME TEMPORARY MEMORY
320         PUMD R30,+R12            !PUSH LENGTH BACK ONTO THE R12 STACK
330         PUMD R26,+R12            !PUSH ADDRESS RETURNED BY 'RSMEM-' ON R12 STACK
340         BIN                      !SET MATH MODE FOR LOOP COUNTER
350         LDB R34,=200             !SET UP MASK
360 MORE    DCM R30                  !DECREMENT LOOP COUNTER
370         JNC DONE                 !JIF NO CHARACTERS LEFT
380         POBD R20,+R36            !GET NEXT CHARACTER
390         ORB R20,R34              !SET MSB OF CURRENT CHARACTER
400         PUBD R20,+R26            !PUSH UNDERLINED CHARACTER BACK
410         JMP MORE                 !GO GET SOME MORE
420 DONE    RTN                      !DONE
430 RSMEM- DAD 37453
440         FIN
```

GRAPHICS CURSOR

GCURS BINARY
Source File: GCURS
Object File: GCURB

This binary program implements a graphics cursor and allows the four cursor keys on the computer to control the cursor. There are five new keywords implemented by the program:

Format: GCURSOR x-location, y-location [, slow-step distance, fast-step distance]

Description: A BASIC statement; x,y is location where cursor is placed on the CRT graphics screen initially. Slow-step distance (optional) is the distance the graphics cursor moves with each press of a cursor control key. Fast-step distance (optional) is the distance the cursor moves with each press of a shifted cursor control key. Default step distances are 1 and 4, respectively.

The cursor keys control the graphics cursor only when a program is running.

Format: GCURSOR OFF

Description: A BASIC statement; turns cursor control keys off and removes the graphics cursor from the CRT screen.

Format: GCURSOR X

Description: A numeric function with no parameters; returns the current x-location of the graphics cursor.

Format: GCURSOR Y

Description: A numeric function with no parameters; returns the current y-location of the graphics cursor.

Format: REV DATE

Description: A string function with no parameters; returns the revision date of the program.

```
1 !****************************
2 !*      GCURS BINARY        *
3 !* (c) Hewlett-Packard Co.  *
4 !*          1980            *
5 !****************************
10         NAM GCURS
20         DEF RUNTIM                !PTR TO RUNTIME ROUTINES TABLE
30         DEF TOKS                  !PTR TO ASCII TABLE
40         DEF PARSES                !PTR TO PARSE ROUTINES TABLE
50         DEF ERMSG                 !PTR TO ERMSG TABLE
60         DEF INIT                  !PTR TO INIT ROUTINE
70 RUNTIM BSZ 2                      !TOK 0 RUNTIME PTR (DUMMY)
80         DEF GCOFF.                !TOK 1 RUNTIME PTR
90         DEF GCURX.                !TOK 2 RUNTIME PTR
100         DEF GCURY.               !TOK 3 RUNTIME PTR
110         DEF GCURS.               !TOK 4 RUNTIME PTR
120         DEF REV.                 !TOK 5 RUNTIME PTR
130 PARSES BSZ 2                     !TOK 0 PARSE ROUTINE (DUMMY)
140         DEF GCOFFP               !  TOK 1 PARSE ROUTINE PTR
150         BSZ 2                    !TOK 2 PARSE ROUTINE (DUMMY)
160         BSZ 2                    !TOK 3 PARSE ROUTINE (DUMMY)
170         DEF GCPAR                !TOK 4 PARSE ROUTINE PTR
180 ERMSG  BYT 377,377               !END OF RELOCATABLE ADDRESSES & ERMSG'S
190 TOKS   ASP "GCURSOR OFF"         !ASCII FOR KEYWORD 1
200        ASP "GCURSOR X"           !ASCII FOR KEYWORD 2
210        ASP "GCURSOR Y"           !ASCII FOR KEYWORD 3
220        ASP "GCURSOR"             !ASCII FOR KEYWORD 4
230        ASP "REV DATE"            !ASCII FOR KEYWORD 5
240        BYT 377                   !END OF KEYWORD TABLE
250 !**********************************************************************
260 INIT   BIN                       !FOR BINARY COMPARE
280        LDBD R34,=ROMFL           !GET ROMFL (REASON FOR INIT)
290        CMB R34,=2                !SCRATCH?
300        JNZ LOAD?                 !JIF NO
310 SCRAT! LDM R44,=236,236,236,236 !LOAD RTNS
320        STMD R44,=CHIDLE          !STORE TO CHIDLE (RETURN HOOK TO SYSTEM)
330        RTN
340 LOAD?  CMB R#,=5                 !LOAD?
350        JZR SCRAT!                !JIF YES, WE'RE GETTING SCRATCHED
360 RTN    RTN                       !DONE, ONLY CASES WE CARE ABOUT
370 !**********************************************************************
380 LEFT   LDMD R40,X14,STEP         !LOAD SLOW STEP OFFSET
390        JMP COMLEF                !GO MOVE CURSOR LEFT
400 RIGHT  LDMD R40,X14,STEP         !LOAD SLOW STEP OFFSET
410        JMP COMRIT                !GO MOVE CURSOR RIGHT
420 KEY    LDMD R14,=BINTAB          !BASE ADDRESS OF BIN PRGM
430        BIN                       !FOR COMPARE
440        CMB R16,=2                !IN RUN MODE?
450        JNZ RTN                   !JIF NO, DON'T DO
460        LDMD R22,=KEYHIT          !GET KEYCODE OF PRESSED KEY
470        CMB R22,=211              !SHIFTED RIGHT CURSOR KEY?
480        JZR FRIGHT                !JIF YES
490        CMB R22,=223              !SHIFTED LEFT CURSOR KEY?
500        JZR FLEFT                 !JIF YES
510        CMB R22,=245              !SHIFTED UP CURSOR KEY?
520        JZR FUP                   !JIF YES
530        CMB R22,=242              !DOWN CURSOR KEY?
540        JZR DOWN                  !JIF YES
550        CMB R22,=234              !LEFT CURSOR KEY?
560        JZR LEFT                  !JIF YES
```

```
570          CMB R22,=235         !RIGHT CURSOR KEY?
580          JZR RIGHT            !JIF YES
590          CMB R22,=241         !UP CURSOR KEY?
600          JZR UP               !JIF YES
610          CMB R22,=254         !SHIFTED DOWN CURSOR KEY?
620          JZR FDOWN            !JIF YES
630          RTN                  !ELSE LET SYSTEM HANDLE THE KEY
640 DOWN     LDMD R40,X14,STEP    !LOAD SLOW STEP CONSTANT
650          JMP COMDOW           !GO MOVE DOWN
660 UP       LDMD R40,X14,STEP    !LOAD SLOW STEP CONSTANT
670          JMP COMUP            !GO MOVE UP
680 FRIGHT LDMD R40,X14,FSTEP     !LOAD FAST STEP CONSTANT
690 COMRIT PUMD R#,+R12           !PUSH STEP VALUE ON R12
700          LDMD R50,X14,CURS-X  !GET CURRENT X FOR ADD
710          PUMD R50,+R12        !PUSH TO R12
720          JSB =ADDROI          !ADD STEP TO CURRENT X
730 COM-X    LDMD R40,X14,CURS-Y  !GET CURRENT Y
740          PUMD R40,+R12        !PUSH TO R12 STACK
750          JMP COMKEY           !GO MOVE CURSOR
760 FLEFT    LDMD R40,X14,FSTEP   !LOAD FAST STEP CONSTANT
770 COMLEF LDMD R50,X14,CURS-X    !GET CURRENT X
780          PUMD R50,+R12        !PUSH FOR SUBTRACT
790          PUMD R40,+R12        !PUSH STEP VALUE FOR SUBTRACT
800          JSB =SUBROI          !SUBTRACT STEP FROM CURRENT X
810          JMP COM-X            !GO PUSH Y AND FINISH
820 FUP      LDMD R40,X14,FSTEP   !LOAD FAST STEP CONSTANT
830 COMUP    LDMD R50,X14,CURS-X  !GET CURRENT X LOCATION
840          PUMD R50,+R12        !PUSH TO R12 STACK
850          PUMD R40,+R12        !PUSH Y-STEP TO R12 STACK
860          LDMD R40,X14,CURS-Y  !GET CURRENT Y LOCATION
870          PUMD R40,+R12        !PUSH TO R12
880          JSB =ADDROI          !ADD STEP TO CURRENT LOCATION
890          JMP COMKEY           !MOVE CURSOR ON SCREEN
900 FDOWN    LDMD R40,X14,FSTEP   !LOAD FAST STEP CONSTANT
910 COMDOW LDMD R50,X14,CURS-X    !GET CURRENT X LOCATION
920          PUMD R50,+R12        !PUSH TO R12 STACK
930          LDMD R50,X14,CURS-Y  !GET CURRENT Y LOCATION
940          PUMD R50,+R12        !PUSH TO R12 STACK
950          PUMD R40,+R12        !PUSH STEP VALUE TO R12
960          JSB =SUBROI          !SUBTRACT STEP VALUE
970 COMKEY JSB X14,PLOT           !ERASE OLD CURSOR
980          CLM R50              !FOR COMPARE
990          POMD R40,-R12        !GET NEW Y
1000          PUMD R40,+R12       !SAVE IT
1010          JSB =COMFLT         !IS Y>=ZERO ?
1020          POMD R40,-R12       !RECOVER Y
1030          JEN TEST-X          !JIF NO
1040          PUMD R40,+R12       !SAVE Y
1050          LDM R50,=2,0,0,0,0,0,20C,19C !REAL 192
1060          JSB =COMFLT         !IS Y<192
1070          POMD R40,-R12       !RECOVER Y
1080          JEZ TEST-X
1090          STMD R40,X14,CURS-Y !STORE IT AWAY
1100 TEST-X CLM R50               !FOR COMPARE
1110          POMD R40,-R12       !GET NEW X
1120          PUMD R40,+R12       !SAVE X
1130          JSB =COMFLT         !X>=0
1140          POMD R40,-R12       !RECOVER X
1150          JEN MOVCUR          !JIF NO
1160          PUMD R40,+R12       !SAVE X
1170          LDM R50,=2,0,0,0,0,0,60C,25C !REAL 256
```

```
1180        JSB =COMFLT          !X<256 ?
1190        BIN                  !COMFLT RETURNS IN BCD MODE
1200        POMD R40,-R12        !RECOVER X
1210        JEZ MOVCUR
1220        STMD R40,X14,CURS-X  !STORE IT AWAY
1230 MOVCUR JSB X14,PLOT         !SPIT OUT NEW CURSOR
1240        CLE                  !FLAG KEY HANDLED
1250        JSB =EOJ2            !RESET R17 & SVCWRD
1260        LDBD R31,X14,KEYCON  !LOAD KEY REPEAT SPEED
1270 LOOPKE LDBD R30,=KEYSTS     !GET KEYBOARD STATUS
1280        LRB R30              !KEY STILL DOWN?
1290        JEV EOJ              !JIF NO
1300        LDBD R30,=CRTSTS     !GET CRT STATUS
1310        LRB R30              !AT RETRACE?
1320        JEV LOOPKE           !JIF YES
1330 LOOPK2 LDBD R30,=KEYSTS     !GET KEYBOARD STATUS
1340        LRB R30              !KEY DEPRESSED?
1350        JEV EOJ              !JIF NO
1360        LDBD R30,=SVCWRD     !ANOTHER KEY?
1370        JOD EOJ              !JIF YES
1380        LDBD R30,=CRTSTS     !GET CRT STATUS
1390        LRB R30              !RETRACE?
1400        JOD LOOPK2           !JIF NO
1410        DCB R31              !DECREMENT WAIT COUNT
1420        JNZ LOOPKE           !JIF NOT DONE WAITING
1430        LDB R31,=KYRPT2      !GET FAST REPEAT COUNT
1440        STBD R31,X14,KEYCON  !SET IN KEYCON FOR FAST REPEAT
1450        LDM R20,=KEY         !GET ADDRESS OF KEY ROUTINE
1460        ADM R20,R14          !MAKE ABSOLUTE (ADD BINTAB)
1470        DCM R20              !DECREMENT FOR LOAD INTO PC
1480        LDM R4,R20           !LOAD PC WITH ADDRESS (DOES A GTO)
1490 EOJ    LDB R31,=KYRPT1      !RESET KEY REPEAT TO SLOW WAIT
1500        STBD R31,X14,KEYCON  !STORE IT
1510        POMD R44,-R6         !THROW AWAY TWO RETURNS
1520        CLE                  !FLAG KEY HANDLED
1530        RTN                  !DONE
1540 !******************************************************************
1550 GCPAR  PUBD R43,+R6         !SAVE INCOMING TOKEN
1560        JSB =NUMVA+          !GET A NUMERIC EXPRESSION
1570        JEN OK               !JIF GOT ONE
1580 ERR    JSB =ERROR+          !ELSE ERROR
1590        BYT 81D              !BAD EXPRESSION
1600 OK     JSB =GETCMA          !DEMAND A COMMA
1610        JSB =NUMVAL          !GET ANOTHER NUMERIC VALUE
1620        JEZ ERR              !JIF NOT THERE
1630        CMB R14,=54          !ANOTHER COMMA?
1640        JNZ DONE             !JIF NO, THAT'S ALL
1650        JSB =NUMVA+          !ELSE GET ANOTHER NUMBER
1660        JEZ ERR              !JIF NOT THERE
1670        JSB =GETCMA          !DEMAND ANOTHER COMMA
1680        JSB =NUMVAL          !GET YET ANOTHER VALUE
1690        JEZ ERR              !JIF NOT THERE
1700 DONE   POBD R47,-R6         !RECOVER INCOMING TOKEN
1710        LDB R45,=371         !LOAD BIN PRGM TOKEN FLAG
1720        PUMD R45,+R12        !PUSH THEM OUT
1730        RTN                  !DONE
1740 GCOFFP PUBD R43,+R6         !SAVE INCOMING TOKEN
1750        JSB =SCAN            !NEED TO DO A SCAN FOR SYSTEM
1760        JMP DONE             !GO FINISH, NO PARAMETERS
1770 !******************************************************************
1780        BYT 241              !ATTRIBUTE(BASIC STAT.,LEGAL AFTER THEN)
```

```
1790 GCOFF. LDMD R14,=BINTAB     !LOAD BASE ADDRESS
1800        JSB X14,SCRAT!       !RELEASE CHIDLE HOOK
1810        JSB X14,PLOT         !ERASE CURSOR
1820        RTN                  !DONE
1830 !*****************************************************************
1840        BYT 0,55
1850 GCURX. LDMD R14,=BINTAB     !GET BASE ADDRESS OF BPGM
1860        LDMD R50,X14,CURS-X  !GET CURRENT X LOCATION
1870 GPUSH  PUMD R50,+R12        !PUSH TO R12 STACK
1880        RTN                  !DONE
1890 !*****************************************************************
1900        BYT 0,55
1910 GCURY. LDMD R14,=BINTAB     !GET BASE ADDRESS
1920        LDMD R50,X14,CURS-Y  !GET CURRENT Y LOCATION
1930        JMP GPUSH            !PUSH TO R12
1940 !*****************************************************************
1950        BYT 241              !ATTRIBUTE(BASIC STAT.,LEGAL AFTER THEN)
1960 GCURS. BIN                  !FOR BINARY MATH
1970        LDMD R14,=BINTAB     !GET BASE ADDRESS
1980        LDM R40,=0,0,0,0,0,0,0,10C !DEFAULT STEP VALUE (1)
1990        STMD R40,X14,STEP
2000        LDB R47,=40C         !DEFAULT FAST STEP VALUE (4)
2010        STMD R40,X14,FSTEP   !STORE IT AWAY
2020        LDM R20,R12          !GET END OF R12 STACK ADDRESS
2030        SBM R20,=40,0        !TRY 4 NUMBERS ON STACK
2040        CMMD R20,=TOS        !YES?
2050        JNZ NOSTEP           !JIF NO STEP VALUES
2060        JSB =ONER            !ELSE GET FAST STEP VALUE
2070        BIN                  !ONER REQUIRES BIN MODE AT ENTRY
2080        STMD R#,X14,FSTEP    !STORE IT AWAY
2090        JSB =ONER            !GET SLOW STEP VALUE
2100        BIN                  !ONER REQUIRES BIN MODE AT ENTRY
2110        STMD R#,X14,STEP     !STORE IT AWAY
2120 NOSTEP JSB =ONER            !GET Y VALUE
2130        BIN                  !ONER REQUIRES BIN MODE AT ENTRY
2140        STMD R#,X14,CURS-Y   !SET CURRENT Y
2150        JSB =ONER            !GET X
2160        BIN                  !ONER RETURNS IN BCD MODE
2170        STMD R#,X14,CURS-X   !SET CURRENT X
2180        JSB X14,PLOT         !OUTPUT CURSOR
2190        LDM R46,=KEY         !GET ADDRESS OF KEY HANDLER ROUTINE
2200        ADM R46,R14          !ADD BASE ADDRESS FOR ABSOLUTE ADDRESS
2210        STM R46,R45          !SET FOR STORE
2220        LDB R47,=236         !LOAD A RTN AFTER IT
2230        LDB R44,=316         !LOAD A JSB IN FRONT
2240        STMD R44,=CHIDLE     !STORE TO CHARCTER IDLE
2250        RTN                  !DONE
2260 !*****************************************************************
2270 PLOT   JSB X14,GCURX.       !PUSH CURRENT X
2280        JSB X14,GCURY.       !PUSH CURRENT Y
2290        LDM R20,=ROMTAB      !GET BASE ADDRESS OF ROM TABLE
2300 NXTROM POMD R24,+R20        !GET NEXT ROM # FROM TABLE
2310        CMB R24,=377         !END OF TABLE?
2320        JZR SYSTEM           !JIF YES, DO SYSTEM MOVE
2330        CMB R24,=PPROM#      !PLOTTER/PRINTER ROM #?
2340        JNZ NXTROM           !JIF NO, TRY NEXT ENTRY
2350        JSB =ROMJSB          !SELECT PLOTTER/PRINTER ROM #
2360        DEF PMOVE.           !JSB TO ITS MOVE ROUTINE
2370        VAL PPROM#           !PLOTTER/PRINTER ROM #
2380        LDMD R14,=BINTAB     !RE-LOAD BPGM BASE ADDRESS
2390        JMP PLOT++           !DO COMMON OUT-CURSOR STUFF
```

```
2400 SYSTEM JSB =MOVE.           !DO A SYSTEM MOVE
2410 PLOT++ LDM R20,=CURSES      !LOAD REL. BASE ADDRESS OF CURSORS
2420        ADM R20,R14          !ADD BPGM BASE FOR ABSOLUTE ADDRESS
2430        LDBD R22,=XMAP       !GET LOWER BYTE OF CRT BIT MAP
2440        ANM R22,=3,0         !KEEP ONLY LOWER TWO BITS
2450        LDM R34,R22          !COPY
2460        LLM R34              !TIMES 2
2470        LLM R34              !TIMES 4
2480        ADM R34,R22          !TIMES 5(EACH CURSOR IS 5 BYTES)
2490        ADM R34,R20          !BASE ADDRESS + OFFSET=CURSOR ADDRESS
2500        LDM R22,=5,0         !LOAD LENGTH OF "STRING"
2510        LDM R44,=1,0,1,0     !LOAD # OF BYTES/LINE AND A COPY
2520        JSB =BPLOT+          !JUMP INTO BPLOT
2530        RTN                  !DONE
2540 !*************************************************************************
2550 CURSES BYT 360,300,240,220,10 !FOUR DIFFERENT CURSORS BECAUSE
2560        BYT 170,140,120,110,4 !BPLOT CAN ONLY WORK TO A FOUR-BIT
2570        BYT 74,60,50,44,2    !RESOLUTION. TO GET 1 BIT RESOLUTION
2580        BYT 36,30,24,22,1    !WE NEED TO USE FOUR DIFFERENT CURSORS
2590 KEYCON BSZ 1                !TEMPORARY KEY REPEAT SPEED
2600 CURS-X BSZ 10               !CURRENT X LOCATION
2610 CURS-Y BSZ 10               !CURRENT Y LOCATION
2620 FSTEP  BSZ 10               !FAST STEP INCREMENT VALUE
2630 STEP   BSZ 10               !SLOW STEP INCREMENT VALUE
2640 !*************************************************************************
2650        BYT 0,56             !ATTRIBUTES(NO PARAM.,$ SYSTEM FUNCTION)
2660 REV.   BIN                  !FOR ADD
2670        LDM R44,=11D,0       !LOAD LEN OF STRING
2680        DEF DATE             !  AND THE RELATIVE ADDRESS
2690        ADMD R46,=BINTAB     !ADD BASE FOR ABSOLUTE ADDRESS
2700        PUMD R44,+R12        !PUSH TO OPERATING STACK
2710        RTN                  !DONE
2720 DATE   ASC "AUG 14,1980"    !DATE STRING
2730 BPLOT+ DAD 34405            !NOTE:
2740 MOVE.  DAD 31703            !MOST OF THESE DEFINITIONS COULD
2750 PMOVE. DAD 64400            !BE REPLACED BY A CALL TO
2760 ROMJSB DAD 4776             !THE GLOBAL FILE
2770 PPROM# EQU 360
2780 ROMTAB DAD 101235
2790 KYRPT2 EQU 1
2800 KYRPT1 EQU 30
2810 CRTSTS DAD 177406
2820 KEYSTS DAD 177402
2830 CHIDLE DAD 102416
2840 ROMFL  DAD 101231
2850 KEYHIT DAD 100671
2860 EOJ2   DAD 34772
2870 ADDROI DAD 52150
2880 SUBROI DAD 52127
2890 BINTAB DAD 101233
2900 NUMVAL DAD 12412
2910 NUMVA+ DAD 12407
2920 SCAN   DAD 11262
2930 GETCMA DAD 13414
2940 SVCWRD DAD 100151
2950 TOS    DAD 101132
2960 ERROR+ DAD 6611
2970 ONER   DAD 56215
2980 XMAP   DAD 100262
2990 COMFLT DAD 32621
3000        FIN                  !END OF SOURCE PROGRAM
```

RECTANGULAR/POLAR CONVERSIONS

RECT/POLAR CONVERSIONS BINARY PROGRAM
Source File: RECPLS
Object File: RECPLB

This program can be used to convert between polar and rectangular coordinates. It implements four BASIC statements:

Format: RECTANGULAR x-variable, y-variable, radius, angle

Description: Sets x- and y-variables equal to the rectangular coordinates that correspond to the specified polar coordinates (radius and angle).

Format: POLAR radius variable, angle variable, x-coordinate, y-coordinate

Description: Sets radius and angle variables equal to the polar coordinates that correspond to the specified x- and y-coordinates.

Format: REV DATE

Description: A string function with no parameters; returns the revision date of the program.

Format: SCRATCHBIN

Description: Scratches the current binary program from computer memory, without affecting anything else.

```
1 !****************************
2 !* RECT/POLAR CONVERSIONS   *
3 !* (c) Hewlett-Packard Co.  *
4 !*          1980            *
5 !****************************
10         NAM R&P                  !SET UP PROGRAM CONTROL BLOCK
20         DEF RUNTIM               !PTR TO RUNTIME ADDRESS TABLE
30         DEF ASCIIS               !PTR TO KEYWORD TABLE
40         DEF PARSE                !PTR TO PARSE ADDRESS TABLE
50         DEF ERMSG                !PTR TO ERROR MESSAGE TABLE
60         DEF INIT                 !PTR TO INIT ROUTINE FOR SYSTEM
70 !**************************************************************************
80 PARSE   BYT 0,0                  !DUMMY TOK #0 PARSE PTR
90         DEF RTPP                 !TOK #1 PARSE PTR
100         DEF RTPP                !TOK #2 PARSE PTR
110         DEF UNLODP              !TOK #3 PARSE PTR
120 RUNTIM BYT 0,0                  !DUMMY TOK #0 RUNTIME
130         DEF RTP.                !TOK #1 RUNTIME
140         DEF PTR.                !TOK #2 RUNTIME
150         DEF SCRB.               !TOK #3 RUNTIME
160         DEF REV.                !TOK #4 RUNTIME
170         BYT 377,377             !TERMINATE RELOCATABLES
180 !**************************************************************************
190 ASCIIS ASP "POLAR"              !KEYWORD #1
200         ASP "RECTANGULAR"       !KEYWORD #2
210         ASP "SCRATCHBIN"        !KEYWORD #3
220         ASP "REV DATE"          !KEYWORD #4
230         BYT 377                 !TERMINATE ASCIIS TABLE
240 !**************************************************************************
250 UNLODP LDB R47,R43              !COPY BPGM TOKEN
260         LDB R45,=371            !LOAD SYSTEM BPGM TOKEN
270         PUMD R45,+R12           !PUSH THE CODE TO THE STACK
280         JSB =SCAN               !SCAN BEFORE RETURNING
290         RTN                     !DONE
300 !**************************************************************************
310 RTPP    PUBD R43,+R6            !SAVE INCOMING TOKEN
320         JSB =SCAN               !SCAN FOR REFNUM
330         JSB =REFNUM             !GET THE 1st VARIABLE REFERENCE
340         JEZ ERR                 !JIF NOT THERE
350         JSB =GETCMA             !DEMAND A COMMA
360         JSB =REFNUM             !GET THE 2nd VARIABLE REFERENCE
370         JEZ ERR                 !JIF NOT THERE
380         JSB =GETCMA             !DEMAND A COMMA
390         JSB =NUMVAL             !GET THE X VALUE
400         JEZ ERR                 !JIF NOT THERE
410         JSB =GETCMA             !DEMAND A COMMA
420         JSB =NUMVAL             !GET THE Y VALUE
430         JEZ ERR                 !JIF NOT THERE
440         POBD R47,-R6            !RECOVER THE INCOMING TOKEN
450         LDB R45,=371            !LOAD THE SYSTEM BPGM TOKEN
460         PUMD R45,+R12           !PUSH THE PARSED CODE
470         RTN                     !DONE
480 !**************************************************************************
490 ERR     POBD R47,-R6            !CLEAN UP R6 (REMOVE TOKEN)
500         JSB =ERROR+             !REPORT ERROR
510         BYT 81D                 !BAD EXPRESSION
520 !**************************************************************************
530 ERMSG   BSZ 0
540         BYT 377                 !NO ERROR MESSAGES
550 !**************************************************************************
```

```
560 INIT    BSZ 0                  !NO INITIALIZATION TO BE DONE
570         RTN                    !DONE
580 !******************************************************************
590 XVAL    BSZ 0
600 RVAL    BSZ 10                 !TEMPORARY STORAGE
610 YVAL    BSZ 0
620 AVAL    BSZ 10                 !TEMPORARY STORAGE
630 !******************************************************************
640         BYT 241                !ATTRIBUTE FOR RECTANGULAR
650 RTP.    JSB =ONER              !GET Y VALUE TO R40
660         LDMD R22,=BINTAB       !LOAD BASE ADDRESS
670         STMD R40,X22,YVAL      !SAVE Y VALUE
680         JSB =ONER              !GET X-VALUE TO R40
690         STMD R40,X22,XVAL      !SAVE X VALUE
700         PUMD R40,+R12          !PUSH FOR MULTIPLY
710         PUMD R40,+R12          !PUSH FOR MULTIPLY
720         JSB =MPYROI            !GET X^2 (LEAVE ON R12)
730         LDMD R40,X22,YVAL      !GET Y VALUE
740         PUMD R40,+R12          !PUSH FOR MULTIPLY
750         PUMD R40,+R12          !PUSH FOR MULTIPLY
760         JSB =MPYROI            !GET Y^2 (LEAVE ON R12)
770         JSB =ADDROI            !GET X^2+Y^2 (LEAVE ON R12)
780         JSB =SQR5              !GET SQR(X^2+Y^2) RADIUS
790         POMD R40,-R12          !RECOVER ANSWER
800         PUMD R40,+R6           !SAVE RESULT FOR LATER
810         LDMD R40,X22,YVAL      !GET Y VALUE
820         PUMD R40,+R12          !PUSH FOR ATN
830         LDMD R40,X22,XVAL      !GET X VALUE
840         PUMD R40,+R12          !PUSH FOR ATN2
850         JSB =ATN2.             !FIND ATN2(Y,X) AND LEAVE ON R12
860         JSB =STOSV             !STORE RESULT TO ANGLE VARIABLE
870         POMD R40,-R6           !RECOVER RADIUS RESULT
880         PUMD R40,+R12          !PUSH FOR STORE
890         JSB =STOSV             !STORE TO THE RADIUS VARIABLE
900         RTN                    !DONE
910 !******************************************************************
920         BYT 241                !ATTRIBUTES FOR POLAR
930 PTR.    JSB =ONER              !GET ANGLE VALUE
940         LDMD R22,=BINTAB       !LOAD BASE ADDRESS
950         STMD R40,X22,AVAL      !STORE FOR LATER
960         JSB =ONER              !GET RADIUS VALUE
970         STMD R40,X22,RVAL      !STORE FOR LATER
980         LDMD R40,X22,AVAL      !GET ANGLE VALUE
990         PUMD R40,+R12          !PUSH FOR COS FUNCTION
1000         JSB =COS10            !TAKE COS(ANGLE)
1010         LDMD R22,=BINTAB      !LOAD BASE ADDRESS
1020         LDMD R40,X22,RVAL     !GET RADIUS VALUE
1030         PUMD R40,+R12         !PUSH FOR MULTIPLY
1040         JSB =MPYROI           !GET R*COS(ANGLE)  X VALUE
1050         POMD R40,-R12         !GET ANSWER
1060         PUMD R40,+R6          !SAVE FOR LATER
1070         LDMD R40,X22,AVAL     !GET ANGLE VALUE
1080         PUMD R40,+R12         !PUSH FOR SIN FUNCTION
1090         JSB =SIN10            !TAKE SIN(ANGLE)
1100         LDMD R22,=BINTAB      !LOAD BASE ADDRESS
1110         LDMD R50,X22,RVAL     !GET RADIUS
1120         PUMD R50,+R12         !PUSH FOR MULTIPLY
1130         JSB =MPYROI           !GET R*SIN(ANGLE) Y VALUE
1140         JSB =STOSV            !STORE TO Y VARIABLE
1150         POMD R40,-R6          !RECOVER X VALUE
1160         PUMD R40,+R12         !PUSH FOR STORE
```

```
1170         JSB =STOSV           !STORE TO X VARIABLE
1180         RTN                  !DONE
1190 !*******************************************************************
1200         BYT 241              !ATTRIBUTES FOR SCRATCHBIN
1210 SCRB.   STBD R#,=GINTDS      !DISABLE INTERRUPTS
1220         LDMD R24,=BINTAB     !LOAD BASE ADDRESS
1230         DCM R24              !MOVE TO LAST BYTE TO KEEP
1240         LDMD R26,=LWAMEM     !GET END OF MEMORY (AND BPGM)
1250         STM R26,R22          !COPY
1260         SBM R22,R24          !GET DISTANCE TO MOVE
1270         LDB R20,=4           !LOAD COUNTER FOR PTR ADJUST
1280         LDM R32,=LAVAIL      !GET ADDRESS OF 1st PTR TO MOVE
1290 UNLD1   LDMD R36,R32         !GET NEXT PTR
1300         ADM R36,R22          !ADD DISTANCE TO MOVE
1310         PUMD R36,+R32        !RESTORE POINTER
1320         DCB R20              !DECREMENT COUNT
1330         JNZ UNLD1            !JIF NOT DONE
1340         LDMD R36,R32         !GET FWBIN
1350         CMMD R36,=LWAMEM     !SAME AS LWAMEM?
1360         JZR UNLD2            !JIF YES
1370         ADM R36,R22          !ELSE ADJUST
1380         STMD R36,R32         !    AND REPLACE
1390 UNLD2   CLM R#
1400         STMD R#,=BINTAB      !ZERO OUT BINTAB (NO BPGM)
1410         LDM R#,R12           !COPY R12 PTR
1420         LDM R41,=316         !LOAD INTO R41-R47 THIS CODE:
1430         DEF MOVDN            !             JSB=MOVDN
1440         STBD R#,=GINTEN      !             STBD R#,=GINTEN
1450         RTN                  !             RTN
1460         STMD R41,R36         !STORE AT END OF R12 STACK
1470         DCM R36              !DCM ADDR. BECAUSE LDM WILL ICM R4 AFTER LOAD
1480         LDM R4,R36           !MOVE PROGRAM EXECUTION TO MOVDN CODE
1490 !*******************************************************************
1500         BYT 0,56             !ATTRIBUTES FOR REV DATE
1510 REV.    LDM R44,=8D,0        !LOAD LENGTH OF STRING
1520         DEF DATE             !    AND RELATIVE ADDRESS OF STRING
1530         ADMD R46,=BINTAB     !MAKE ADDRESS ABSOLUTE
1540         PUMD R44,+R12        !PUSH TO STACK
1550         RTN                  !DONE
1560 DATE    ASC "05/05/80"
1570 !*******************************************************************
1580 COS10   DAD 53556
1590 MPYROI  DAD 52722
1600 ADDROI  DAD 52150
1610 SIN10   DAD 53546
1620 SQR5    DAD 52442
1630 ATN2.   DAD 76455
1640 ONER    DAD 56215
1650 ERROR+  DAD 06611
1660 NUMVAL  DAD 12412
1670 GETCMA  DAD 13414            !DEFINE ADDRESSES
1680 REFNUM  DAD 17025
1690 SCAN    DAD 11262
1700 STOSV   DAD 45254
1710 BINTAB  DAD 101233
1720 GINTDS  DAD 177401
1730 LWAMEM  DAD 100022
1740 LAVAIL  DAD 100010
1750 MOVDN   DAD 37324
1760 GINTEN  DAD 177400
1770         FIN
```

RECTANGULAR/POLAR CONVERSIONS (ROM)

RECT/POLAR CONVERSIONS
ROM VERSION
Source File: ROMPRS
Object File: ROMPRB

This program is the same as the RECT/POLAR CONVERSIONS binary program, except that it is written for a ROM.

```
10 !**************************
20 !* RECT/POLAR CONVERSIONS  *
30 !*      ROM VERSION        *
40 !* (c) Hewlett-Packard Co. *
50 !*         1980            *
60 !**************************
70        ABS ROM 60000
80        BYT 100                !ROM # MUST BE FIRST BYTE
90        BYT 277                !ROM COMPLEMENT # MUST BE SECOND BYTE
100        DEF RUNTIM            !PTR TO RUNTIME ADDRESS TABLE
110        DEF ASCIIS            !PTR TO KEYWORD TABLE
120        DEF PARSE             !PTR TO PARSE ADDRESS TABLE
130        DEF ERMSG             !PTR TO ERROR MESSAGE TABLE
140        DEF INIT              !PTR TO INIT ROUTINE FOR SYSTEM
150 !*********************************************************************
160 PARSE  BYT 0,0               !DUMMY TOK #0 PARSE PTR
170        DEF RTPP              !TOK #1 PARSE PTR
180        DEF RTPP              !TOK #2 PARSE PTR
190        DEF UNLODP            !TOK #3 PARSE PTR
200 RUNTIM BYT 0,0               !DUMMY TOK #0 RUNTIME
210        DEF RTP.              !TOK #1 RUNTIME
220        DEF PTR.              !TOK #2 RUNTIME
230        DEF SCRB.             !TOK #3 RUNTIME
240        DEF REV.              !TOK #4 RUNTIME
250 !*********************************************************************
260 ASCIIS ASP "POLAR"           !KEYWORD #1
270        ASP "RECTANGULAR"     !KEYWORD #2
280        ASP "SCRATCHBIN"      !KEYWORD #3
290        ASP "REV DATE"        !KEYWORD #4
300        BYT 377               !TERMINATE ASCIIS TABLE
310 !*********************************************************************
320 UNLODP LDM R46,=370,100      !SYSTEM EXTERNAL ROM TOKEN & ROM #
330        PUMD R46,+R12         !PUSH THEM TO THE STACK
340        PUBD R43,+R12         !PUSH INCOMING TOKEN TO THE STACK
350        JSB =ROMJSB           !MUST CALL THROUGH ROMJSB
360        DEF SCAN              !CALL SCAN FOR SYSTEM
370        BYT 0                 !IT'S IN ROM 0
380        RTN                   !DONE
390 !*********************************************************************
400 RTPP   PUBD R43,+R6          !SAVE INCOMING TOKEN
410        JSB =ROMJSB           !SCAN SELECTS OTHER ROMS
420        DEF SCAN              !DO A SCAN FOR REFNUM
430        BYT 0                 !SELECT ROM 0
440        JSB =ROMJSB
450        DEF REFNUM            !GET THE 1rst VARIABLE REFERENCE
460        BYT 0                 !ROM #0
470        JEZ ERR               !JIF NOT THERE
480        JSB =ROMJSB
490        DEF GETCMA            !DEMAND A COMMA
500        BYT 0                 !ROM #0
510        JSB =ROMJSB
520        DEF REFNUM            !GET THE 2nd VARIABLE REFERENCE
530        BYT 0
540        JEZ ERR               !JIF NOT THERE
550        JSB =ROMJSB
560        DEF GETCMA            !DEMAND A COMMA
570        BYT 0
580        JSB =ROMJSB
590        DEF NUMVAL            !GET THE X VALUE
```

```
600        BYT 0
610        JEZ ERR              !JIF NOT THERE
620        JSB =ROMJSB
630        DEF GETCMA           !DEMAND A COMMA
640        BYT 0
650        JSB =ROMJSB
660        DEF NUMVAL           !GET THE Y VALUE
670        BYT 0
680        JEZ ERR              !JIF NOT THERE
690        POBD R47,-R6         !RECOVER THE INCOMING TOKEN
700        LDB R46,=100         !LOAD THE ROM #
710        LDB R45,=370         !LOAD THE SYSTEM BPGM TOKEN
720        PUMD R45,+R12        !PUSH THE PARSED CODE
730        JMP GTOROM           !DONE
740 !*****************************************************************
750 ERR    POBD R47,-R6         !CLEAN UP R6 (REMOVE TOKEN)
760        JSB =ERROR           !REPORT ERROR
770        BYT 81D              !BAD EXPRESSION
780 GTOROM GTO ROMRTN           !HAVE TO RESELECT ROM 0 WHEN RETURNING FROM PARS
E
790 !*****************************************************************
800 ERMSG  BSZ 0
810        BYT 377              !NO ERROR MESSAGES
820 !*****************************************************************
830 INIT   BSZ 0
840        BIN
850        LDBD R34,=ROMFL      !GET REASON FOR INIT
860        JNZ INIRTN           !JIF NOT POWER ON
870        LDMD R34,=FWUSER     !GET FIRST AVAILABLE WORD
880        STMD R34,=UNBAS1     !SAVE BASE ADDRESS
890        ADM R34,=20,0        !PLUS # OF BYTES NEEDED
900        STMD R34,=FWUSER     !RESET FIRST WORD AVAILABLE PTR
910        JSB =ROMJSB
920        DEF SCRAT+           !RE-SET UP THE BASIC PROGRAM STRUCTURE AND PTRS
930        BYT 0
940 INIRTN RTN
950 !*****************************************************************
960 XVAL   EQU 0
970 RVAL   EQU 0                !INDEX INTO STOLEN RAM
980 YVAL   EQU 10
990 AVAL   EQU 10               !INDEX INTO STOLEN RAM
1000 !****************************************************************
1010        BYT 241             !ATTRIBUTE FOR RECTANGULAR
1020 RTP.   JSB =ONER           !GET Y VALUE TO R40
1030        LDMD R22,=UNBAS1    !LOAD BASE ADDRESS
1040        STMD R40,X22,YVAL   !SAVE Y VALUE
1050        JSB =ONER           !GET X-VALUE TO R40
1060        STMD R40,X22,XVAL   !SAVE X VALUE
1070        PUMD R40,+R12       !PUSH FOR MULTIPLY
1080        PUMD R40,+R12       !PUSH FOR MULTIPLY
1090        JSB =MPYROI         !GET X^2 (LEAVE ON R12)
1100        LDMD R40,X22,YVAL   !GET Y VALUE
1110        PUMD R40,+R12       !PUSH FOR MULTIPLY
1120        PUMD R40,+R12       !PUSH FOR MULTIPLY
1130        JSB =MPYROI         !GET Y^2 (LEAVE ON R12)
1140        JSB =ADDROI         !GET X^2+Y^2 (LEAVE ON R12)
1150        JSB =SQR5           !GET SQR(X^2+Y^2) RADIUS
1160        POMD R40,-R12       !RECOVER ANSWER
1170        PUMD R40,+R6        !SAVE RESULT FOR LATER
1180        LDMD R40,X22,YVAL   !GET Y VALUE
```

```
1190        PUMD R40,+R12       !PUSH FOR ATN
1200        LDMD R40,X22,XVAL   !GET X VALUE
1210        PUMD R40,+R12       !PUSH FOR ATN2
1220        JSB =ROMJSB         !FIND ATN2(Y,X) AND LEAVE ON R12
1222        DEF ATN2.
1224        BYT 0
1230        JSB =ROMJSB         !STORE RESULT TO ANGLE VARIABLE
1232        DEF STOSV
1234        BYT 0
1240        POMD R40,-R6        !RECOVER RADIUS RESULT
1250        PUMD R40,+R12       !PUSH FOR STORE
1260        JSB =ROMJSB         !STORE TO THE RADIUS VARIABLE
1262        DEF STOSV
1264        BYT 0
1270        RTN                 !DONE
1280 !*********************************************************************
1290        BYT 241             !ATTRIBUTES FOR POLAR
1300 PTR.   JSB =ONER           !GET ANGLE VALUE
1310        LDMD R22,=UNBAS1    !LOAD BASE ADDRESS
1320        STMD R40,X22,AVAL   !STORE FOR LATER
1330        JSB =ONER           !GET RADIUS VALUE
1340        STMD R40,X22,RVAL   !STORE FOR LATER
1350        LDMD R40,X22,AVAL   !GET ANGLE VALUE
1360        PUMD R40,+R12       !PUSH FOR COS FUNCTION
1370        JSB =COS10          !TAKE COS(ANGLE)
1380        LDMD R22,=UNBAS1    !LOAD BASE ADDRESS
1390        LDMD R40,X22,RVAL   !GET RADIUS VALUE
1400        PUMD R40,+R12       !PUSH FOR MULTIPLY
1410        JSB =MPYROI         !GET R*COS(ANGLE)  X VALUE
1420        POMD R40,-R12       !GET ANSWER
1430        PUMD R40,+R6        !SAVE FOR LATER
1440        LDMD R40,X22,AVAL   !GET ANGLE VALUE
1450        PUMD R40,+R12       !PUSH FOR SIN FUNCTION
1460        JSB =SIN10          !TAKE SIN(ANGLE)
1470        LDMD R22,=UNBAS1    !LOAD BASE ADDRESS
1480        LDMD R50,X22,RVAL   !GET RADIUS
1490        PUMD R50,+R12       !PUSH FOR MULTIPLY
1500        JSB =MPYROI         !GET R*SIN(ANGLE) Y VALUE
1510        JSB =ROMJSB         !STORE TO Y VARIABLE
1512        DEF STOSV
1514        BYT 0
1520        POMD R40,-R6        !RECOVER X VALUE
1530        PUMD R40,+R12       !PUSH FOR STORE
1540        JSB =ROMJSB         !STORE TO X VARIABLE
1542        DEF STOSV
1544        BYT 0
1550        RTN                 !DONE
1560 !*********************************************************************
1570        BYT 241             !ATTRIBUTES FOR SCRATCHBIN
1580 SCRB.  LDMD R24,=BINTAB    !GET BASE ADDRESS OF BINARY
1582        JZR SCRTN           !JIF NONE PRESENT
1590        STBD R24,=GINTDS    !DISABLE GLOBAL INTERRUPTS
1595        BIN
1600        DCM R24             !MOVE TO LAST BYTE TO KEEP
1610        LDMD R26,=LWAMEM    !GET END OF MEMORY (AND BPGM)
1620        STM R26,R22         !COPY
1630        SBM R22,R24         !GET DISTANCE TO MOVE
1640        LDB R20,=4          !LOAD COUNTER FOR PTR ADJUST
1650        LDM R32,=LAVAIL     !GET ADDRESS OF 1rst PTR TO MOVE
1660 UNLD1  LDMD R36,R32        !GET NEXT PTR
```

```
1670        ADM R36,R22         !ADD DISTANCE TO MOVE
1680        PUMD R36,+R32       !RESTORE POINTER
1690        DCB R20             !DECREMENT COUNT
1700        JNZ UNLD1           !JIF NOT DONE
1710        LDMD R36,R32        !GET FWBIN
1720        CMMD R36,=LWAMEM    !SAME AS LWAMEM?
1730        JZR UNLD2           !JIF YES
1740        ADM R36,R22         !ELSE ADJUST
1750        STMD R36,R32        !    AND REPLACE
1760 UNLD2  CLM R#
1770        STMD R#,=BINTAB     !ZERO OUT BINTAB (NO BPGM)
1780        JSB =MOVDN          !MOVE MEMORY TO HIGHER ADDRESS
1790        STBD R#,=GINTEN     !RE-ENABLE INTERRUPTS
1800 SCRTN  RTN                 !DONE
1810 !*****************************************************************
1820        BYT 0,56            !ATTRIBUTES FOR REV DATE
1830 REV.   LDM R44,=8D,0       !LOAD LENGTH OF STRING
1840        DEF DATE            !     AND ADDRESS OF STRING
1850        PUMD R44,+R12       !PUSH TO STACK
1860        RTN                 !DONE
1870 DATE   ASC "05/05/80"
1880 !*****************************************************************
1890 COS10  DAD 53556
1900 MPYROI DAD 52722
1910 ADDROI DAD 52150
1920 SIN10  DAD 53546
1930 SQR5   DAD 52442
1940 ATN2   DAD 76455
1950 ONER   DAD 56215
1960 ERROR  DAD 06615
1970 NUMVAL DAD 12412
1980 GETCMA DAD 13414           !DEFINE ADDRESSES
1990 REFNUM DAD 17025
2000 SCAN   DAD 11262
2010 STOSV  DAD 45254
2020 BINTAB DAD 101233
2030 GINTDS DAD 177401
2040 LWAMEM DAD 100022
2050 LAVAIL DAD 100010
2060 MOVDN  DAD 37324
2070 GINTEN DAD 177400
2080 ROMJSB DAD 4776
2090 FWUSER DAD 100000
2100 UNBAS1 DAD 102554
2110 ROMRTN DAD 4762
2120 SCRAT+ DAD 4344
2130 ROMFL  DAD 101231
2140        FIN
```

NOTES

SECTION 9

THE HP-82928A SYSTEM MONITOR

The HP-82928A System Monitor is an optional plug-in module for use with the HP-83/85 Assembler ROM. The System Monitor:

--Permits the user to set two breakpoints in any portion of memory. Any time either of these two addresses is referenced in any manner, an interrupt is caused. The user can use this interrupt to examine CPU registers, status bits, and memory locations, and to make changes, if desired.

--Permits the user to single-step and trace through the operation of code at any point in memory.

The System Monitor may be used only in conjunction with the HP-83/85 Assembler ROM.

SETTING AND CLEARING BREAKPOINTS

Two System Monitor commands, BKP and CLR, permit the user to set and clear breakpoints.

BKP

System Monitor Command

Set Breakpoint

Format: BKP octal address [, select code for output]

Description: Sets breakpoint (BP) #1 or #2 at the specified address in HP-83/85 memory. If no breakpoints are set, the command sets BP1. If BP1 is already set, the command sets BP2. If BP1 and BP2 are both set, the command resets BP2 to the new octal address; BP1 remains set at its original address. Breakpoints can be set at any address in HP-83/85 system RAM or ROM. Breakpoints can be cleared only by the CLR command.

When execution is halted at a breakpoint, the B key is a typing aid for BKP.

When the address at which a breakpoint is set is encountered during execution of a program or a calculator mode statement, execution halts and a block of status information is output to the device specified by the select code. If no select code is specified, the default is 1 (CRT IS device) at power-on, or the last select code specified by a breakpoint.

The information output comprises the following:

Memory Contents: The contents of a specified number of RAM or ROM locations are output. The output is based on the specifications in the last MEM statement or command, if one was previously executed. Output begins with the octal address specified in the last-executed MEM and continues for the number of bytes specified by that last MEM.

If no MEM was executed, the default address is 0; default number of bytes is 100_8.

Output can be generated from a ROM, as specified by the ROM# in the MEM last executed. Default ROM# is 0.

Like MEM, the output first shows the octal values of the quantities in the block of memory, eight bytes to a line of output, then shows the ASCII representation of the quantities.

<u>CPU Status Indicators</u>: This output includes the following:

PC: The setting of the program counter (i.e., the contents of CPU registers R4 and R5). When execution is resumed, it will begin at the address specified by PC.

AR: Contents of the address register pointer (i.e., the current AR).

DR: Contents of the data register pointer (i.e., the current DR).

BKPS: Addresses of breakpoints BP1 and BP2. An address of 000000 can mean no breakpoint is set or a breakpoint is set at address 000000.

OV: Status of overflow flag.

CY: Status of carry flag.

NG: Status of MSB (most significant bit), used to indicate a negative quantity.

LZ: Status of LDZ (left digit zero) flag.

ZR: Status of Z (zero) flag.

RZ: Status of RDZ (right digit zero) flag.

OD: Status of LSB (least significant bit), used to indicate an odd quantity.

DC: Setting of DCM (decimal) flag.

E: Contents of E (extend) register. This will be a quantity between 0 and 17_8.

<u>CPU Registers</u>: Octal contents of all CPU registers, eight bytes to a line of output.

Once a breakpoint has been encountered and execution is halted, the following keys on the keyboard are active for the uses shown:

Key	Use
B	Typing aid for BKP command.
C	Typing aid for CLR command.
M	Typing aid for MEM command.
P	Typing aid for PC= command.
R	Typing aid for REG command.
T	Typing aid for TRACE command.
[STEP]	Single-step execution.
[ROLL ▲]	Roll up display.
[ROLL ▼]	Roll down display.
[RUN]	Resume normal program execution.
[BACK SPACE]	
[COPY]	
[PAPER ADVANCE]	

Most other keys on the keyboard are inactive at a breakpoint, although once the entry of a system monitor command has been begun, all alphanumeric keys are once again active to allow the full command to be entered.

Example: Here is a sample of a breakpoint output.

Memory Contents (octal)

```
MEM 0
026 000 112 205 155 071 112 205
345 074 106 075 065 075 044 075
070 205 123 205 123 205 106 251
300 202 230 136 262 001 377 251
340 037 262 030 377 321 000 140
366 012 262 231 202 261 014 140
036 306 000 000 316 322 007 316
055 072 230 316 034 205 117 220
```

Memory Contents (ASCII)

```
  J m9J e<F=5=$=
8 S S F)@  ^2 )
' 2 Q 'v 2  1 '
 F  NR N-: N  0
MEM 0
```

PC and Breakpoint Status

```
PC      DR AR BKPS
003160 74 20 003157 000000
OV CY NG LZ ZR RZ OD DC E
0  0  0  1  0  0  1  1  00
```

CPU Register Contents

```
REG
000 000 227 141 160 006 304 202
320 211 325 211 015 001 001 001
157 006 231 251 321 211 316 211
321 212 040 000 107 211 001 000
015 000 000 000 000 231 251 002
116 000 040 000 200 003 000 000
040 040 040 040 040 040 176 003
001 004 000 000 000 000 000 000
```

The contents of memory and CPU registers are shown with eight succeeding registers per row; thus, the top row of the CPU register output shows registers RØ-R7, the second row R10-R17, etc.

CLR

System Monitor Command

Clear Breakpoint

Format:		
	CLR 1	Clears BP1
	CLR 2	Clears BP2
	CLR [any number except 1 or 2]	Clears BP1 and BP2

Description: Clears breakpoint #1, breakpoint #2, or both breakpoints.

After a breakpoint has halted execution, C is a typing aid for CLR.

After CLR is displayed, the user can type 1 [END LINE] to clear BP1 or 2 [END LINE] to clear BP2. After CLR is displayed, simply pressing [END LINE] or entering any number except 1 or 2, then pressing [END LINE], clears both BP1 and BP2.

CLR may be used any time execution has been halted, whether or not it has been halted by a breakpoint.

OPERATIONS AT A BREAKPOINT

After execution has halted after a breakpoint, the user can:

--Generate an output of the contents of a specified number of bytes of memory.
--Change the program counter.
--Change contents of any CPU register.
--Perform single-step and TRACE execution.
--Use [ROLL] or [SHIFT] [ROLL] to examine the CRT screen.
--Use [RUN] to resume normal execution, beginning with the memory byte currently addressed by the program counter (PC).

MEM

System Monitor Command

Memory Dump to CRT

Format: MEM address [: ROM#] [, # of bytes] [= #, #, ...]

Description: Acts like Assembler-provided BASIC statement MEM, except that at a breakpoint M acts as a typing aid for MEM.

PC=

System Monitor Command

Program Counter Is

Format: PC= address between 0 and 177377

Description: Changes contents of program counter (CPU registers R4 and R5) to the specified address, and dumps CPU status and memory contents exactly as when a breakpoint (BKP) is executed.

After a breakpoint has been executed, P acts as a typing aid for PC=.

When execution is resumed, it will begin at the address now specified by the contents of the program counter.

This command is active only after execution has been halted by a breakpoint.

Example: PC = 3477 Sets the PC to resume execution with byte 003477.

REG

System Monitor Command

CPU Register Is

Format: REG number of CPU register = value between 0 and octal 377

Description: Changes contents of specified CPU register to specified value, and dumps CPU status and memory contents exactly as when a breakpoint (BKP) is executed. Value may be specified as octal, decimal, or BCD quantity. This command is active only after execution has been halted by a breakpoint. R acts as a typing aid for REG.

Example:

REG 35 = 31 Changes contents of register R34 to 31_8.

REG 36 = 19C Changes contents of register R36 to BCD 19.

REG 37 = 25D Changes contents of register R37 to 25_{10}.

STEP

System Monitor Command

Single-Step Execution

Format: This command is executed with the [STEP] key.

Description: Executes the next complete machine code instruction (not merely the next byte), beginning with the location currently addressed by the PC, then halts and dumps CPU status and memory contents exactly as when a breakpoint (BKP) is executed. Active only after execution has been halted by a breakpoint.

TRACE

System Monitor Command

Trace Execution

Format: TRACE octal, decimal or BCD value

Description: Resumes execution with the next machine code instruction, and continues for the number of instructions (not bytes) specified by the octal, decimal or BCD value.

After each instruction is executed, CPU breakpoint and partial CPU status is output to the current CRT IS device. When execution halts, the CPU status and memory contents are output as at a breakpoint.

The information output after each instruction comprises the following:

PC: The current setting of the program counter (i.e., the contents of CPU registers R4 and R5).

DR: Current data register.

AR: Current address register.

BKPS: Addresses of breakpoints BP1 and BP2. (Because of the internal coding of the System Monitor, the address of BP1 appears to increase as each instruction is traced and status is output. However, when trace execution halts, both breakpoints are reset to their original addresses when the TRACE command was executed.)

The information output when execution halts after tracing is exactly the same as that output at a breakpoint: that is, the contents of the memory block specified by the last MEM statement or command, complete CPU status, and the contents of all CPU registers. See System Monitor command BKP for details.

Example: TRACE 10 Generates an output similar to the following:

Tracing PC, DR, AR, BP1, BP2

```
003161 74 12 003160 000000
003162 74 12 003161 000000
003163 36 12 003162 000000
003164 36 12 003163 000000
003165 36 76 003164 000000
003166 36 76 003165 000000
003167 76 76 003166 000000
```

Memory Contents (octal)

```
MEM 0
026 000 112 205 155 071 112 205
345 074 106 075 065 075 044 075
070 205 123 205 123 205 106 251
300 202 230 136 262 001 377 251
340 037 262 030 377 321 000 140
366 012 262 231 202 261 014 140
036 306 000 000 316 322 007 316
055 072 230 316 034 205 117 220
```

Memory Contents (ASCII)

```
  J m9J e<F=5=$=
8 S S F)@   ^2 )
` 2 Q `v 2  1 `
 F  NR N-: N  O
MEM 0
```

CPU and Breakpoint Status

```
PC      DR AR BKPS
003170 76 76 003157 000000
OV CY NG LZ ZR RZ OD DC E
0  0  0  1  1  1  0  1  00
```

CPU Register Contents (octal)

```
REG
000 000 077 211 170 006 304 202
320 211 321 211 015 001 001 001
157 006 231 251 321 211 316 211
321 212 040 000 107 211 160 000
015 000 000 000 000 231 251 002
116 000 040 000 200 003 000 000
040 040 040 040 040 040 176 003
001 004 000 000 001 000 000 000
```

APPENDIX A

GLOSSARY OF TERMS

Allocated program. Form of program where variable space has been allocated, variable names are addresses, and line references have become addresses. An allocated program is ready to run, and cannot be edited.

BASIC reserved word. Entry in an ASCII table. From the user's point of view, a BASIC reserved word is an entry that has meaning for the system: it can be entered as a command, statement, or function. From the system point of view, a BASIC reserved word is the decompiled form of a token.

Binary program. Assembly-language program which can be loaded into the HP-83 or HP-85 and run. A binary program should be relocatable.

Calculator mode statement. Contains BASIC statements as well as numeric or string operations. Compare to expression.

Command. Non-programmable language element. Commands are executed immediately; they cannot be used in a program. With the Assembler ROM installed there are two types of commands:

--System command. Available in normal BASIC mode; these commands may or may not be available in Assembler mode (e.g., COPY, SCRATCH).

--Assembler command. Available only in Assembler mode (e.g., BASIC, ALOAD).

Deallocated program. Form of input text rendered into tokens. Deallocated program contains actual variable names and immediate data, and can be edited.

Effective address. Location of the ultimate, fully-computed address or destination of an instruction.

Expression. Contains purely numeric or string operations. Compare to calculator mode statement.

Glossary of Terms

Function. Programmable BASIC language element that can be used as part of a statement. A function, such as PI, SIN, ABS, etc., always returns a value.

HP-83/85. Applies to either HP-83 or HP-85 Personal Computer.

Instruction. Programmable assembly language element. These are of two types:

--CPU instruction. Instructions for the machine central processing unit.

--Pseudo-instruction. Instructions to the Assembler ROM at assembly time.

Label. Identifier that corresponds to an address or value.

Object code. The assembled machine code for a binary or ROM program. Object code is ready to be run.

PC. Program counter in computer CPU hardware.

PCR. System program counter, controlled by software.

ROM program. Assembly-language program which can be burned into a ROM package for later connection to and running on the HP-83/85. A ROM program is not relocatable.

Source code. Instructions and pseudo-instructions before assembly, as they are entered from the keyboard.

Statement. Programmable BASIC language element. A statement does not return a value and cannot be used in an expression.

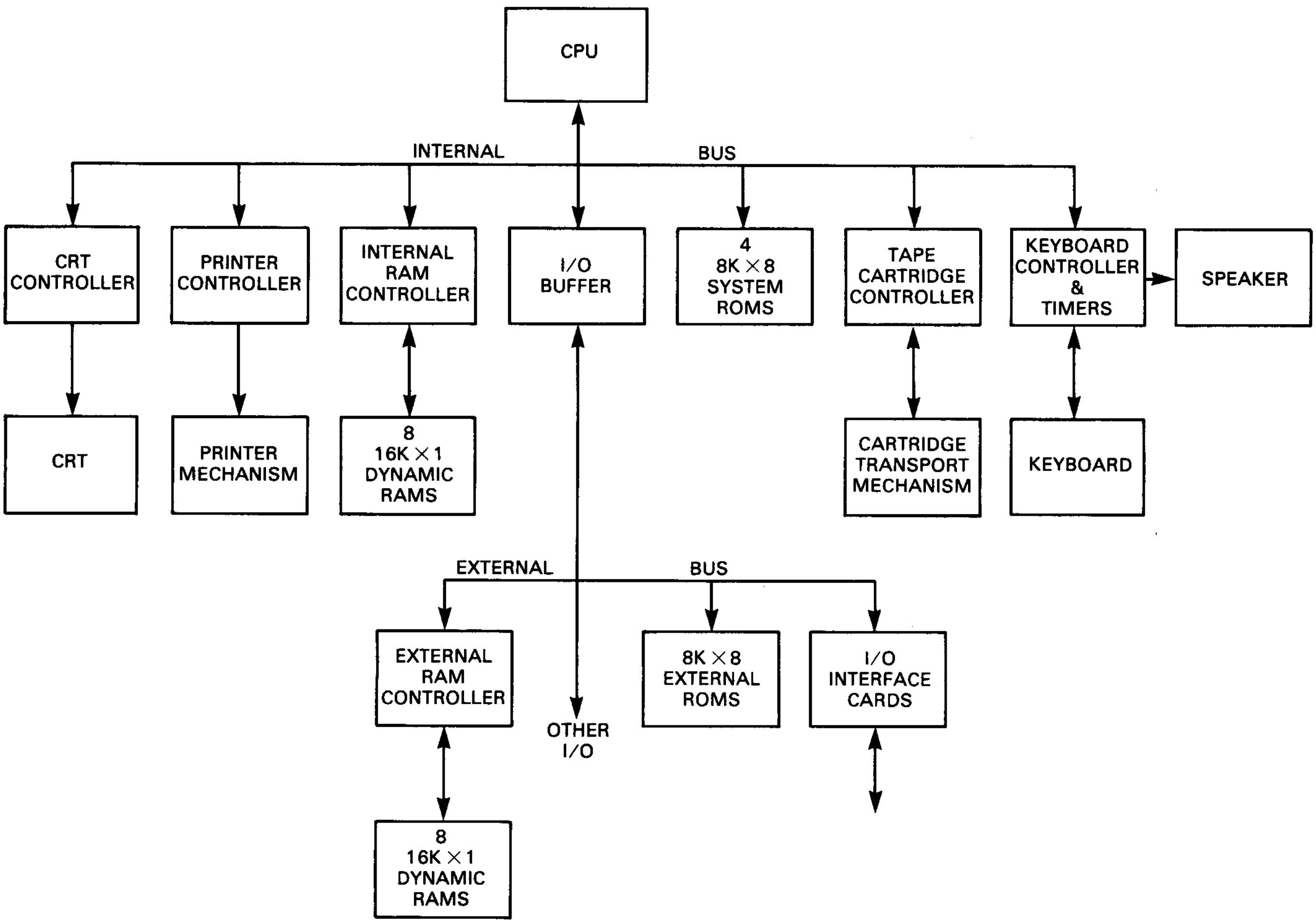
Token: A one-byte numeric quantity representing a keyword. A token indicates to the machine the addresses of the ASCII entry, runtime routine, and parse routine (possibly implied) associated with the keyword. Each token also has associated methods of allocation, deallocation, parsing, and decompiling.

Variable. A numeric value which may be assigned to a label. Variables can be simple numeric, array, or string; if numeric, they can be real, short, or integer.

NOTES

APPENDIX B

SYSTEM HARDWARE DIAGRAM



NOTES

APPENDIX C

ASSEMBLER INSTRUCTION SET

On the following pages is a list of all CPU instructions available on the Assembler ROM.

LEGEND

DR	Data register. Can be register number (e.g., R32), R* or R#.
AR	Address register. Can be register number (e.g., R32), R* or R#.
<u>Literal</u>	Literal value, up to 10_8 bytes in length. Can be BCD constant (e.g., 99C), octal constant (e.g., 12), or decimal constant (e.g., 20D). Can also be specified by a label, where the literal quantity is a one- or two-byte value or address assigned to the label.
<u>Label</u>	Address of literal quantity. Label name must begin with an alphabetic character, can use any combination of alphanumeric characters, and can be 1-6 characters in length.
Clock Cycle	1.6 μsec.
B	Number of bytes.
T	Add one clock cycle if true (i.e., the jump occurs).
R(x)	CPU register addressed by (x).
M(x)	Memory location addressed by (x). (x) must be a 16-bit address.
PC	Program Counter. CPU registers R4 and R5. Used to address the instruction being executed.

SP	Subroutine Stack Pointer. CPU registers R6 and R7. Used to point to the next available location on the subroutine return address stack.
EA	Effective Address. The location from which data is read for load-type instructions or the location where data is placed for store-type instructions.
ADR	Address. The two-byte quantity directly following an instruction that uses the literal direct, literal indirect, index direct or index indirect addressing mode. This quantity is always an address.
n	Literal value.
←	Is transferred to.
( )	Contents of.
———	Complement (e.g., $\overline{x}$ is complement of x). This is one's complement if DCM=0 and nine's complement if DCM=1.
•	Logical AND.
V	Inclusive OR.
⊕	Exclusive OR.
JIF	Jump if.
1	Status bit is set.
Ø	Status bit is cleared.
X	Status bit is affected.

-	Status bit is not affected.
Y	This option is available to this instruction.

The complete list of CPU instructions begins on the next page.

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status											Binary/ BCD Option
											DCM=Ø			DCM=1			
						LSB	MSB	RDZ LDZ	Z	DCM	E	CY	OVF	E	CY	OVF	
ADB DR, AR	Add byte	Reg. imm.	302	5	DR←DR+AR	X	X	X	X	-	-	X	X	-	X	0	Y
ADB DR, = literal	Add byte	Lit. imm.	312	5	DR←DR+M(PC+1)	X	X	X	X	-	-	X	X	-	X	0	Y
ADBD DR, AR	Add byte	Reg. dir.	332	6	DR←DR+M(AR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADBD DR, = label	Add byte	Lit. dir.	322	5	DR←DR+M(ADR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADM DR, AR	Add multi-byte	Reg. imm.	303	4+B	DR←DR+AR	X	X	X	X	-	-	X	X	-	X	0	Y
ADM DR, = literal	Add multi-byte	Lit. imm.	313	4+B	DR←DR+M(PC+1)	X	X	X	X	-	-	X	X	-	X	0	Y
ADMD DR, AR	Add multi-byte	Reg. dir.	333	5+B	DR←DR+M(AR)	X	X	X	X	-	-	X	X	-	X	0	Y
ADMD DR, = label	Add multi-byte	Lit. dir.	323	4+B	DR←DR+M(ADR)	X	X	X	X	-	-	X	X	-	X	0	Y
ANM DR, AR	Logical AND (multi-byte)	Reg. imm.	307	4+B	DR←DR·AR	X	X	X	X	-	-	0	0	-	0	0	
ANM DR, = literal	Logical AND (multi-byte)	Lit. imm.	317	4+B	DR←DR·M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
ANMD DR, AR	Logical AND (multi-byte)	Reg. Dir.	337	5+B	DR←DR·M(AR)	X	X	X	X	-	-	0	0	-	0	0	
ANMD DR, = literal	Logical AND (multi-byte)	Lit. dir.	327	5+B	DR←DR·M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
ARP AR	Load ARP		000-077 (≠001)	2	ARP←n	-	-	-	-	-	-	-	-	-	-	-	
ARP *	Load ARP with contents of RØ		001	3	ARP←RØ	-	-	-	-	-	-	-	-	-	-	-	
BCD	Set BCD mode		231	4	DCM←1	-	-	-	-	1	-	-	-	-	-	-	
BIN	Set binary mode		230	4	DCM←0	-	-	-	-	0	-	-	-	-	-	-	
CLB DR	Clear byte	Reg. imm.	222	5	DR←0	X	X	X	X	-	-	0	0	-	0	0	
CLM DR	Clear multi-byte	Reg. imm.	223	4+B	DR←0	X	X	X	X	-	-	0	0	-	0	0	
CLE	Clear E		235	2	E←0	-	-	-	-	-	0	-	-	0	-	-	
CMB DR, AR	Compare byte	Reg. imm.	300	5	DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status: LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø: E	DCM=Ø: CY	DCM=Ø: OVF	DCM=1: E	DCM=1: CY	DCM=1: OVF	Binary/ BCD Option
CMB DR, = literal	Compare byte	Lit. imm.	310	5	DR+$\overline{M(PC+1)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMBD DR, AR	Compare byte	Reg. dir.	330	6	DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMBD DR, = label	Compare byte	Lit. dir.	320	6	DR+$\overline{M(ADR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMM DR, AR	Compare multi-byte	Reg. imm.	301	4+B	DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMM DR, = literal	Compare multi-byte	Lit. imm.	311	4+B	DR+$\overline{M(PC+1)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMMD DR, AR	Compare multi-byte	Reg. dir.	331	5+B	DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
CMMD DR, = label	Compare multi-byte	Lit. dir.	321	5+B	DR+$\overline{M(ADR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
DCB DR	Decrement byte	Reg. imm.	212	5	DR←DR-1	X	X	X	X	-	-	X	X	-	X	0	Y
DCM DR	Decrement multi-byte	Reg. imm.	213	4+B	DR←DR-1	X	X	X	X	-	-	X	X	-	X	0	Y
DCE	Decrement E		233	2	E←E-1	-	-	-	-	-	X	-	-	X	-	-	
DRP DR	Load DRP		100-177 (≠101)	2	DRP←n	-	-	-	-	-	-	-	-	-	-	-	
DRP 1	Load DRP with contents of RØ		101	3	DRP←RØ	-	-	-	-	-	-	-	-	-	-	-	
ELB DR	Extended left byte	Reg. imm.	200	5	Circulate DR left once	X	X	X	X	-	-	X	X	X	0	0	Y
ELM DR	Extended left multi-byte	Reg. imm.	201	4+B	Circulate DR left once	X	X	X	X	-	-	X	X	X	0	0	Y
ERB DR	Extended right byte	Reg. imm.	202	5	Circulate DR right once	X	X	X	X	-	-	X	0	X	0	0	Y
ERM DR	Extended right multi-byte	Reg. imm.	203	4+B	Circulate DR right once	X	X	X	X	-	-	X	0	X	0	0	Y
ICB DR	Increment byte	Reg. imm.	210	5	DR←DR+1	X	X	X	X	-	-	X	X	-	X	0	Y
ICM DR	Increment multi-byte	Reg. imm.	211	4+B	DR←DR+1	X	X	X	X	-	-	X	X	-	X	0	Y

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status											Binary/BCD Option
						LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	
ICE	Increment E		234	2	E←E+1	-	-	-	-	-	X	-	-	X	-	-	
JCY label	Jump on carry		373	4+T	JIF←CY=1	-	-	-	-	-	-	-	-	-	-	-	
JEN label	Jump on E non-zero		370	4+T	JIF E≠0000	-	-	-	-	-	-	-	-	-	-	-	
JEV label	Jump on even		363	4+T	JIF LSB=0	-	-	-	-	-	-	-	-	-	-	-	
JEZ label	Jump on E zero		371	4+T	JIF E=0000	-	-	-	-	-	-	-	-	-	-	-	
JLN label	Jump on left digit non-zero		375	4+T	JIF LDZ≠1	-	-	-	-	-	-	-	-	-	-	-	
JLZ label	Jump on left digit zero		374	4+T	JIF LDZ=1	-	-	-	-	-	-	-	-	-	-	-	
JMP label	Unconditional jump		360	4+T	Jump always	-	-	-	-	-	-	-	-	-	-	-	
JNC label	Jump on no carry		372	4+T	JIF CY=0	-	-	-	-	-	-	-	-	-	-	-	
JNG label	Jump on negative		364	4+T	JIF MSB≠OVF	-	-	-	-	-	-	-	-	-	-	-	
JNO label	Jump on no overflow		361	4+T	JIF OVF=0	-	-	-	-	-	-	-	-	-	-	-	
JNZ label	Jump on non-zero		366	4+T	JIF Z≠1	-	-	-	-	-	-	-	-	-	-	-	
JOD label	Jump on odd		362	4+T	JIF LSB=1	-	-	-	-	-	-	-	-	-	-	-	
JPS label	Jump on positive		365	4+T	JIF MSB=OVF	-	-	-	-	-	-	-	-	-	-	-	
JRN label	Jump on right digit non-zero		377	4+T	JIF RDZ≠1	-	-	-	-	-	-	-	-	-	-	-	
JRZ label	Jump on right digit zero		376	4+T	JIF RDZ=1	-	-	-	-	-	-	-	-	-	-	-	
JSB=label	Jump subroutine	Literal direct	316	9	Jump subroutine	-	-	-	-	-	-	-	-	-	-	-	
JSB XR, label	Jump subroutine	Indexed	306	11	Jump subroutine indexed	-	-	-	-	-	-	-	-	-	-	-	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=0 E	DCM=0 CY	DCM=0 OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
JZR label	Jump on zero		367	4+T	JIF Z=1	-	-	-	-	-	-	-	-	-	-	-	
LDB DR, AR	Load byte	Reg. imm.	240	5	DR←AR	X	X	X	X	-	-	0	0	-	0	0	
LDB DR, = literal	Load byte	Lit. imm.	250	5	DR←M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, AR	Load byte	Reg. dir.	244	6	DR←M(AR)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, = label	Load byte	Lit. dir.	260	6	DR←M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
LDBD DR, XAR, label	Load byte	Index dir.	264	8	DR←M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, AR	Load byte	Reg-indir.	254	8	DR←M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, = label	Load byte	Lit. indir.	270	8	DR←M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
LDBI DR, XAR, label	Load byte	Index indir.	274	10	DR←M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
LDM DR, AR	Load multi-byte	Reg. imm.	241	4+B	DR←AR	X	X	X	X	-	-	0	0	-	0	0	
LDM DR, = literal	Load multi-byte	Lit. imm.	251	4+B	DR←M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, AR	Load multi-byte	Reg. dir.	245	5+B	DR←M(AR)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, = label	Load multi-byte	Lit. dir.	261	5+B	DR←M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
LDMD DR, XAR, label	Load multi-byte	Index dir.	265	7+B	DR←M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, AR	Load multi-byte	Reg. indir.	255	7+B	DR←M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, = label	Load multi-byte	Lit. indir.	271	7+B	DR←M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
LDMI DR, XAR, label	Load multi-byte	Index indir.	275	9+B	DR←M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	DCM=Ø CY	DCM=Ø OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
LLB DR	Logical left byte	Reg. imm.	204	5	Logical left shift DR	X	X	X	X	-	-	X	X	X	0	0	Y
LLM DR	Logical left multi-byte	Reg. imm.	205	4+B	Logical left shift DR	X	X	X	X	-	-	X	X	X	0	0	Y
LRB DR	Logical right byte	Reg. imm.	206	5	Logical right shift DR	X	X	X	X	-	-	X	0	X	0	0	Y
LRM DR	Logical right multi-byte	Re. imm.	207	4+B	Logical right shift DR	X	X	X	X	-	-	X	0	X	0	0	Y
NCB DR	Nine's (or one's) complement byte	Reg. imm.	216	5	DR←$\overline{DR}$	X	X	X	X	-	-	X	X	-	X	0	Y
NCM DR	Nine's (or one's) complement multi-byte	Reg. imm.	217	4+B	DR←$\overline{DR}$	X	X	X	X	-	-	X	X	-	X	0	Y
ORB DR, AR	Or byte inclusive	Reg. imm.	224	5	DR←DR∨AR	X	X	X	X	-	-	0	0	-	0	0	
ORM DR, AR	Or multi-byte inclusive	Reg. imm.	225	4+B	DR←DR∨AR	X	X	X	X	-	-	0	0	-	0	0	
PAD	Pop ARP, DRP and status from stack		237	8	Status←M(SP)	X	X	X	X	X	-	X	X	-	X	X	
POBD DR,+AR	Pop byte with post-increment	Stk. dir.	340	6	DR←M(AR), AR←AR+1	X	X	X	X	-	-	0	0	-	0	0	
POBD DR,-AR	Pop byte with with pre-decrement	Stk. dir.	342	6	DR←M(AR), AR←AR-1	X	X	X	X	-	-	0	0	-	0	0	
POBI DR,+AR	Pop byte with post-increment	Stk. indir.	350	8	DR←M(M(AR)), AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
POBI DR,-AR	Pop byte with pre-decrement	Stk. indir.	352	8	DR←M(M(AR)), AR←AR-2	X	X	X	X	-	-	0	0	-	0	0	
POMD DR,+AR	Pop multi-byte with post-increment	Stk. dir.	341	5+B	DR←M(AR), AR←AR+M	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=Ø E	CY	OVF	DCM=1 E	CY	OVF	Binary/ BCD Option
POMD DR,-AR	Pop multi-byte with pre-decrement	Stk. dir.	343	5+B	DR←M(AR), AR←AR-M	X	X	X	X	-	-	0	0	-	0	0	
POMI DR,+AR	Pop multi-byte with post-increment	Stk. indir.	351	7+B	DR←M(M(AR)), AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
POMI DR,-AR	Pop multi-byte with pre-decrement	Stk. indir.	353	7+B	DR←M(M(AR)), AR←AR-2	X	X	X	X	-	-	0	0	-	0	0	
PUBD DR,+AR	Push byte with post-increment	Stk. dir.	344	6	M(AR)←DR, AR←AR+1	X	X	X	X	-	-	0	0	-	0	0	
PUBD DR,-AR	Push byte with pre-decrement	Stk. dir.	346	6	AR←AR-1, M(AR)←DR	X	X	X	X	-	-	0	0	-	0	0	
PUBI DR,+AR	Push byte with post-increment	Stk. indir.	354	8	M(M(AR))←DR, AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
PUBI DR,-AR	Push byte with pre-decrement	Stk. indir.	356	8	AR←AR-2, M(M(AR))←DR	X	X	X	X	-	-	0	0	-	0	0	
PUMD DR,+AR	Push multi-byte with post-increment	Stk. dir.	345	5+B	M(AR)←DR, AR←AR+M	X	X	X	X	-	-	0	0	-	0	0	
PUMD DR,-AR	Push multi-byte with pre-decrement	Stk. dir.	347	5+B	AR←AR-M, M(AR)←DR	X	X	X	X	-	-	0	0	-	0	0	
PUMI DR,+AR	Push multi-byte with post-increment	Stk. indir.	355	7+B	M(M(AR))←DR, AR←AR+2	X	X	X	X	-	-	0	0	-	0	0	
PUMI DR,-AR	Push multi-byte with pre-decrement	Stk. indir.	357	7+B	AR←AR-2, M(M(AR))←DR	X	X	X	X	-	-	0	0	-	0	0	
RTN	Subroutine return		236	5	SP←SP-2, PC←M(SP)	-	-	-	-	-	-	-	-	-	-	-	
SAD	Save ARP, DRP and status on stack		232	8	M(SP)←Status	-	-	-	-	-	-	-	-	-	-	-	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status LSB	MSB	RDZ LDZ	Z	DCM	DCM=0 E	DCM=0 CY	DCM=0 OVF	DCM=1 E	DCM=1 CY	DCM=1 OVF	Binary/ BCD Option
SBB DR, AR	Subtract byte	Reg. imm.	304	5	DR←DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBB DR, = literal	Subtract byte	Lit. imm.	314	5	DR←DR+$\overline{M(PC+1)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBBD DR, AR	Subtract byte	Reg. dir.	334	6	DR←DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBBD DR, = label	Subtract byte	Lit. dir.	324	6	DR←DR+$\overline{M(ADR)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBM DR, AR	Subtract multi-byte	Reg. imm.	305	4+B	DR←DR+$\overline{AR}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBM DR, = literal	Subtract multi-byte	Lit. imm.	315	4+B	DR←DR+$\overline{M(PC+1)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
SBMD DR, AR	Subtract multi-byte	Reg. dir.	335	5+B	DR←DR+$\overline{M(AR)}$+1	X	X	X	X	-	-	X	X	-	X	0	Y
SBMD DR, = literal	Subtract multi-byte	Lit. dir.	325	5+B	DR←DR+$\overline{M(ADR)}$ +1	X	X	X	X	-	-	X	X	-	X	0	Y
STB DR, AR	Store byte	Reg. imm.	242	5	DR→AR	X	X	X	X	-	-	0	0	-	0	0	
STB DR, = literal	Store byte	Lit. imm.	252	5	DR→M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, AR	Store byte	Reg. dir.	246	6	DR→M(AR)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, = label	Store byte	Lit. dir.	262	6	DR→M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
STBD DR, XAR, label	Store byte	Index dir.	266	8	DR→M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, AR	Store byte	Reg. indir.	256	8	DR→M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, = label	Store byte	Lit. indir.	272	8	DR→M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
STBI DR, XAR, label	Store byte	Index indir	276	10	DR→M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
STM DR, AR	Store multi-byte	Reg. imm.	243	4+B	DR→AR	X	X	X	X	-	-	0	0	-	0	0	
STM DR, = literal	Store multi-byte	Lit. imm.	253	4+B	DR→M(PC+1)	X	X	X	X	-	-	0	0	-	0	0	
STMD DR, AR	Store multi byte	Reg. dir.	247	5+B	DR→M(AR)	X	X	X	X	-	-	0	0	-	0	0	

Instruction Format	Description	Addressing Mode	OpCode	Clock Cycles	Operation	Status											Binary/ BCD Option
											DCM=Ø			DCM=1			
						LSB	MSB	RDZ LDZ	Z	DCM	E	CY	OVF	E	CY	OVF	
STMD DR, = label	Store multi-byte	Lit. dir.	263	5+B	DR→M(ADR)	X	X	X	X	-	-	0	0	-	0	0	
STMD DR, XAR, label	Store multi-byte	Index dir.	267	7+B	DR→M(ADR+AR)	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, AR	Store multi-byte	Reg. indir.	257	7+B	DR→M(M(AR))	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, = label	Store multi-byte	Lit. indir.	273	7+B	DR→M(M(ADR))	X	X	X	X	-	-	0	0	-	0	0	
STMI DR, XAR, label	Store multi-byte	Index indir	277	9+B	DR→M(M(ADR+ AR))	X	X	X	X	-	-	0	0	-	0	0	
TCB DR	Ten's (or two's) complement byte	Reg. imm.	214	5	DR←$\overline{DR}$+1	X	X	X	X	-	-	0	0	-	0	0	Y
TCM DR	Ten's (or two's) complement multi-byte	Reg. imm.	215	4+B	DR←$\overline{DR}$+1	X	X	X	X	-	-	0	0	-	0	0	Y
TSB DR	Test byte	Reg. imm.	220	5	Test DR	X	X	X	X	-	-	X	X	-	X	0	Y
TSM DR	Test multi-byte	Reg. imm.	221	4+B	Test DR	X	X	X	X	-	-	X	X	-	X	0	Y
XRB DR, AR	Or byte exclusive	Reg. imm.	226	5	DR←DR ⊕ AR	X	X	X	X	-	-	0	0	-	0	0	
XRM DR, AR	Or multi-byte exclusive	Reg. imm.	227	4+B	DR←DR ⊕ AR	X	X	X	X	-	-	0	0	-	0	0	

NOTES

Appendix D

ASSEMBLER INSTRUCTION CODING

The chart below shows how the CPU instructions appear when assembled into machine language object code by the computer.

7	6	5	4	3	2	1	0
0	DRP/ ARP	≠000001 Load with literal =000001 Load with RØ					
1	0	0	0	0	Logical/ Extended	Right/Left	M/B
1	0	0	0	1	0	Decrement/ Increment	M/B
1	0	0	0	1	1	Nine's Complement/ Ten's Complement	M/B
1	0	0	1	0	0	Clear/Test	M/B
1	0	0	1	0	1	XOR/OR	M/B
1	0	0	1	1	000 BIN 001 BCD 010 SAD 011 DCE 100 ICE 101 CLE 110 RTN 111 PAD		
1	0	1	000 REG IMM 001 REG DIR 010 LIT IMM 011 REG IND 100 LIT DIR 101 INX DIR 110 LIT IND 111 INX IND			Store/Load	M/B
1	1	0	00 REG IMM 01 LIT IMM 10 LIT DIR 11 REG DIR		00 CMP 01 ADD 10 SUB 11 AND		M/B 1
1	1	0	00 INX 01 LIT		11 JSB		0
1	1	1	0	IND/ DIR	PUSH/ POP	-ADR/ +ADR	M/B
1	1	1	1		000 001 010 011 100 101 110 111		JNO/JMP JEV/JOD JPS/JNG JZR/JNZ JEZ/JEN JCY/JNC JLN/JLZ JRN/JRZ

X/Y = 1/0

NOTES

Appendix E

ASCII TABLE

The following is a table of all the ASCII keycodes on the HP-83/85.

NOTE

The keycodes used in the HP-83/85 are very close to, but in some cases not exactly the same as, ASCII codes.

KEYCODE DEC	KEYCODE OCT	ASCII CHR	KEY	KEYCODE DEC	KEYCODE OCT	ASCII CHR	KEY
0	0	◀	ctrl @	47	57	/	/
1	1	¿	ctrl A	48	60	0	0
2	2	x̄	ctrl B	49	61	1	1
3	3	N̄	ctrl C	50	62	2	2
4	4	α	ctrl D	51	63	3	3
5	5	β	ctrl E	52	64	4	4
6	6	Γ	ctrl F	53	65	5	5
7	7	π̄	ctrl G	54	66	6	6
8	10	Δ	ctrl H	55	67	7	7
9	11	σ	ctrl I	56	70	8	8
10	12	↑	ctrl J	57	71	9	9
11	13	λ	ctrl K	58	72	:	:
12	14	μ	ctrl L	59	73	;	;
13	15		ctrl M	60	74	<	<
14	16	τ	ctrl N	61	75	=	=
15	17	Φ	ctrl O	62	76	>	>
16	20	Θ	ctrl P	63	77	?	?
17	21	Ω	ctrl Q	64	100	@	@
18	22	δ	ctrl R	65	101	A	A
19	23	Â	ctrl S	66	102	B	B
20	24	â	ctrl T	67	103	C	C
21	25	Ä	ctrl U	68	104	D	D
22	26	ä	ctrl V	69	105	E	E
23	27	Ö	ctrl W	70	106	F	F
24	30	ö	ctrl X	71	107	G	G
25	31	Ü	ctrl Y	72	110	H	H
26	32	ü	ctrl Z	73	111	I	I
27	33	Æ	ctrl [	74	112	J	J
28	34	æ	ctrl \	75	113	K	K
29	35	≠	ctrl]	76	114	L	L
30	36	£	ctrl ^	77	115	M	M
31	37	▒	ctrl _	78	116	N	N
32	40		SPACE	79	117	O	O
33	41	!	!	80	120	P	P
34	42	"	"	81	121	Q	Q
35	43	#	#	82	122	R	R
36	44	$	$	83	123	S	S
37	45	%	%	84	124	T	T
38	46	&	&	85	125	U	U
39	47	'	'	86	126	V	V
40	50	(	(	87	127	W	W
41	51	)	)	88	130	X	X
42	52	*	*	89	131	Y	Y
43	53	+	+	90	132	Z	Z
44	54	,	,	91	133	[	[
45	55	-	-	92	134	\	\
46	56	.	.	93	135	]	]

KEYCODE DEC	KEYCODE OCT	ASCII CHR	KEY	KEYCODE DEC	KEYCODE OCT	ASCII CHR	KEY
94	136	^	^	161	241	!	UP CURSOR
95	137	_	_	162	242	"	DOWN CURS
96	140	`	s KEY LABEL	163	243	#	INS/RPL
97	141	a	a	164	244	$	DEL CHR
98	142	b	b	165	245	%	HOME CURS
99	143	c	c	166	246	&	RESULT
100	144	d	d	167	247	'	
101	145	e	e	168	250	(	DELETE
102	146	f	f	169	251	)	STORE
103	147	g	g	170	252	*	LOAD
104	150	h	h	171	253	+	
105	151	i	i	172	254	,	AUTO
106	152	j	j	173	255	-	SCRATCH
107	153	k	k	174	256	.	
108	154	l	l	175	257	/	
109	155	m	m	176	260	0	
110	156	n	n	177	261	1	
111	157	o	o	178	262	2	
112	160	p	p	179	263	3	
113	161	q	q	180	264	4	
114	162	r	r	181	265	5	
115	163	s	s	182	266	6	
116	164	t	t	183	267	7	
117	165	u	u	184	270	8	
118	166	v	v	185	271	9	
119	167	w	w	186	272	:	
120	170	x	x	187	273	;	
121	171	y	y	188	274	<	
122	172	z	z	189	275	=	
123	173	π	s /	190	276	>	
124	174	¦	¦	191	277	?	
125	175	→	s −	192	300	@	
126	176	Σ	s *	193	301	A	
127	177	⊢	s +	194	302	B	
128	200	◀	K1	195	303	C	
129	201	¿	K2	196	304	D	
130	202	Ẍ	K3	197	305	E	
131	203	Ñ	K4	198	306	F	
132	204	α	K5	199	307	G	
133	205	β	K6	200	310	H	
134	206	Γ	K7	201	311	I	
135	207	ō	K8	202	312	J	
136	210	Δ	REW	203	313	K	
137	211	σ	COPY	204	314	L	
138	212	↑	PAPER ADV	205	315	M	
139	213	λ	RESET	206	316	N	
140	214	µ	INIT	207	317	O	
141	215	_	RUN	208	320	P	
142	216	τ	PAUSE	209	321	Q	
143	217	Φ	CONT	210	322	R	
144	220	θ	STEP	211	323	S	
145	221	Ω	TEST	212	324	T	
146	222	δ	CLR SCREEN	213	325	U	
147	223	Å	GRAPH	214	326	V	
148	224	å	LIST	215	327	W	
149	225	Ä	PLIST	216	330	X	
150	226	ä	KEY LABEL	217	331	Y	
151	227	Ö		218	332	Z	
152	230	ö		219	333	[	
153	231	Ü	BACKSPACE	220	334	\	
154	232	ü	END LINE	221	335	]	
155	233	Æ	FAST BCKSP	222	336	^	
156	234	æ	LEFT CURS	223	337	_	
157	235	≠	RIGHT CURS	224	340	`	
158	236	£	ROLL UP	225	341	a	
159	237	▒	ROLL DOWN	226	342	b	
160	240		CLR LINE	227	343	c	

KEYCODE DEC	KEYCODE OCT	ASCII CHR	KEY
228	344	d	
229	345	e	
230	346	f	
231	347	g	
232	350	h	
233	351	i	
234	352	j	
235	353	k	
236	354	l	
237	355	m	
238	356	n	
239	357	o	
240	360	p	
241	361	q	
242	362	r	
243	363	s	
244	364	t	
245	365	u	
246	366	v	
247	367	w	
248	370	x	
249	371	y	
250	372	z	
251	373	π	
252	374	¦	
253	375	÷	
254	376	Σ	
255	377	⊏	

NOTES

Appendix F

TABLE OF TOKENS AND ATTRIBUTES

The following is a table of the system tokens and attributes used in the HP-83 and HP-85.

	ROUTINE	NAME	TOKEN	ATTRIB
TAB.R	DEF ERRORX	ERROR	0	0,44
	DEF FTSVL	SNV	1	0,1
	DEF SVADR	SAV	2	0,1
	DEF FTSTL	STRVAR	3	0,1
	DEF ICONST	REAL CONST	4	0,4
	DEF SCONST	"QUOTED STR	5	0,5
	DEF SCONST	UNQUOT STR	6	0,5
	DEF STOST	STO STRING	7	0,31
	DEF STOSV	STORE SV	10	0,31
	DEF AVADR1	1-DIM ADR	11	0,32
	DEF AVADR2	2-DIM ADR	12	0,32
	DEF AVVAL1	1-DIM VALUE	13	0,32
	DEF AVVAL2	2-DIM VALUE	14	0,32
	DEF ERRORX	CARRIAGE RTN	15	0,44
	DEF GORTN	ENDSTMT	16	0,0
	DEF ERRORX	DUMMY	17	0,44
	DEF ERRORX	DUMMY	20	0,44
	DEF FTADR	SNV ADR	21	0,3
	DEF SVADR+	SAV ADR	22	0,3
	DEF FTSTLS	SAVE STR	23	0,3
	DEF STOSVM	MULTI STO	24	0,43
	DEF STOSTM	MULTI STO$	25	0,43
	DEF FNCAL.	FUNCTION CL	26	0,6
	DEF FNCAL$	STR FUNC CL	27	0,6
	DEF JTRUE#	JMP TRUE	30	0,7
	DEF ERRORE	ILLEGAL END	31	0,44
	DEF INTCON	INT CONST	32	0,2
	DEF JFALSR	JMP FALSE	33	0,11
	DEF JMPREL	JMP REL	34	0,26
	DEF SUBST1	1 DIM SUBST	35	0,34
	DEF SUBST2	2 DIM SUBST	36	0,34
	DEF EJMP#	ELSE J#	37	0,25
	DEF ERRORX	DUMMY	40	0,44
	DEF ERRORX	DUMMY	41	0,44
	DEF P#ARAY	Array PRINT#	42	0,36
	DEF ERRORX	DUMMY	43	0,44
	DEF R#ARAY	Array READ#	44	0,44
	DEF ERRORX	:	45	0,44
	DEF CONCA.	& CONCAT	46	7,53
	DEF NOP47.	;	47	0,42
	DEF ERRORX	(	50	0,44
	DEF ERRORX	)	51	0,44
	DEF MPYROI	*	52	12,51
	DEF ADDROI	+	53	7,51
	DEF ERRORX	,	54	0,44
	DEF SUBROI	- DIADIC	55	7,51
	DEF ERRORX	.	56	0,44

DEF DIV2	/	57	12,51
DEF YTX5	^	60	14,51
DEF UNEQ$.	#	61	6,53
DEF LEQ$.	<=	62	6,53
DEF GEQ$.	>=	63	6,53
DEF UNEQ$.	<>	64	6,53
DEF EQ$.	=	65	6,53
DEF GR$.	>	66	6,53
DEF LT$.	<	67	6,53
DEF CHSROI	- MONADIC	70	7,50
DEF UNEQ.	#	71	6,51
DEF LEQ.	<=	72	6,51
DEF GEQ.	>=	73	6,51
DEF UNEQ.	<>	74	6,51
DEF EQ.	=	75	6,51
DEF GR.	>	76	6,51
DEF LT.	<	77	6,51
DEF ATSIGN	@	100	0,42
DEF ONERR.	ON ERROR	101	0,241
DEF OFFER.	OFF ERROR	102	0,241
DEF ONKEY.	ON KEY#	103	0,241
DEF OFKEY.	OFF KEY#	104	0,241
DEF AUTO.	AUTO	105	0,141
DEF BEEP.	BEEP	106	0,241
DEF CLEAR.	CLEAR	107	0,241
DEF CONTI.	CONT	110	0,141
DEF ONTIM.	ON TIMER#	111	0,241
DEF INIT.	INIT	112	0,141
DEF LIST.	LIST	113	0,241
DEF BPLOT.	BPLOT	114	0,241
DEF STIME.	SETTIME	115	0,241
DEF ERRORX	ERROR	116	0,44
DEF ERRORX	ERROR	117	0,44
DEF READ#.	READ#	120	0,241
DEF RENAM.	RENAME	121	0,241
DEF ALPHA.	ALPHA	122	0,241
DEF CRT.	CRT IS	123	0,241
DEF RUN.	RUN	124	0,141
DEF DEG.	DEG	125	0,241
DEF DISP.	DISP	126	0,241
DEF GCLR.	GCLEAR	127	0,241
DEF SCRAT.	SCRATCH	130	0,141
DEF DEFA+.	DEFAULT ON	131	0,241
DEF JMPLN#	GOTO	132	0,210
DEF JMPSUB	GOSUB	133	0,210
DEF PRNT#.	PRINT #	134	0,241
DEF GRAD.	GRAD	135	0,241
DEF GRAPH.	GRAPH	136	0,241
DEF INPUT.	INPUT	137	0,241
DEF IDRAW.	IDRAW	140	0,241
DEF FNLET.	LET FN	141	0,217
DEF NOP.	LET	142	0,241
DEF PRALL.	PRINT ALL	143	0,241
DEF CAT.	CAT	144	0,241

DEF DRAW.	DRAW	145	0,241
DEF ON.	ON	146	0,230
DEF LABEL.	LABEL	147	0,241
DEF WAIT.	WAIT	150	0,241
DEF PLOT.	PLOT	151	0,241
DEF PRINS.	PRINTER IS	152	0,241
DEF PRINT.	PRINT	153	0,241
DEF RAD.	RAD	154	0,241
DEF RNDIZ.	RANDOMIZE	155	0,241
DEF READ.	READ	156	0,241
DEF STORB.	STORE BIN	157	0,241
DEF RESTO.	RESTORE	160	0,241
DEF RETRN.	RETURN	161	0,241
DEF OFTIM.	OFF TIMER#	162	0,241
DEF MOVE.	MOVE	163	0,241
DEF FLIP.	FLIP	164	0,241
DEF STOP.	STOP	165	0,241
DEF ERRORX	ERROR	166	0,44
DEF PENUP.	PENUP	167	0,241
DEF TRCVB.	TRACE VRBL	170	0,241
DEF TRCAL.	TRACE ALL	171	0,241
DEF XAXIS.	XAXIS	172	0,241
DEF YAXIS.	YAXIS	173	0,241
DEF COPY.	COPY	174	0,241
DEF NORMA.	NORMAL	175	0,241
DEF ERAST.	ERASE TAPE	176	0,241
DEF SKIPI	INTEGER	177	0,323
DEF SKIPS	SHORT	200	0,322
DEF DELET.	DELETE	201	0,141
DEF SCALE.	SCALE	202	0,241
DEF SKIP!	REMARK	203	0,241
DEF OPTIO.	OPTION BASE	204	0,315
DEF SKIPC	COM	205	0,324
DEF SKIPEM	DATA	206	0,320
DEF SKPDEF	DEF FN	207	0,312
DEF SKIPD	DIM	210	0,321
DEF KEYLA.	KEY LABEL	211	0,241
DEF STOP.	END	212	0,241
DEF FNRTN.	FN END	213	0,313
DEF FOR.	FOR	214	0,341
DEF ERRORT	IF	215	0,344
DEF SKIPIT	IMAGE	216	0,341
DEF NEXT.	NEXT	217	0,341
DEF ERRORX	ERROR	220	0,44
DEF ERRORT	LET (IMPLY)	221	0,244
DEF ASIGN.	ASSIGN	222	0,241
DEF CREAT.	CREATE	223	0,241
DEF PURGE.	PURGE	224	0,241
DEF REWIN.	REWIND	225	0,241
DEF LOADB.	LOADBIN	226	0,241
DEF PAUSE.	PAUSE	227	0,241
DEF ERRORX	ERROR	230	0,44
DEF SKIPR	REAL	231	0,321
DEF RENUM.	REN	232	0,141

Table of Tokens and Attributes

	DEF	SKIP!	!	233	0,241
	DEF	DEFA-.	DEFAULT OFF	234	0,241
	DEF	PEN.	PEN	235	0,241
	DEF	PLIST.	PLIST	236	0,241
	DEF	LDIR.	LDIR	237	0,241
	DEF	IMOVE.	IMOVE	240	0,241
	DEF	FNLET.	FN ILET	241	0,217
	DEF	CTAPE.	CTAPE	242	0,241
	DEF	TRACE.	TRACE	243	0,241
	DEF	TO.	TO	244	0,41
	DEF	OR.	OR	245	2,51
	DEF	MAX10	MAX	246	40,55
	DEF	TIME.	TIME	247	0,55
	DEF	DATE.	DATE	250	0,55
	DEF	FP5	FP	251	20,55
	DEF	IP5	IP	252	20,55
	DEF	EPS10	EPSILON	253	0,55
	DEF	REM10	RMD	254	40,55
	DEF	CEIL10	CEIL	255	20,55
	DEF	ATN2.	ATN(X/Y)	256	40,55
	DEF	ERRORX	DUMMY	257	0,44
	DEF	SQR5	SQR	260	20,55
	DEF	MIN10	MIN	261	40,55
	DEF	ERRORX	DUMMY	262	0,44
	DEF	ABS5	ABS	263	20,55
	DEF	ICOS	ACS	264	20,55
	DEF	ISIN	ASN	265	20,55
	DEF	ITAN	ATN	266	20,55
	DEF	SGN5	SGN	267	20,55
	DEF	ERRORX	DUMMY	270	0,44
	DEF	COT10	COT	271	20,55
	DEF	CSEC10	CSC	272	20,55
	DEF	ERRORX	DUMMY	273	0,44
	DEF	EXP5	EXP	274	20,55
	DEF	INT5	INT	275	20,55
	DEF	LOGT5	LGT (10)	276	20,55
ASICS	DEF	LN5	LOG (E)	277	20,55
	DEF	ERRORX	DUMMY	300	0,44
	DEF	SEC10	SEC	301	20,55
	DEF	CHR$.	CHR$	302	20,56
	DEF	VAL$.	VAL$	303	20,56
	DEF	LEN.	LEN	304	30,55
	DEF	NUM.	NUM	305	30,55
	DEF	VAL.	VAL	306	30,55
	DEF	INF10	INF	307	0,55
	DEF	RND10	RND	310	0,55
	DEF	PI10	PI	311	0,55
	DEF	UPC$.	UPC$	312	30,56
	DEF	USING.	USING	313	0,341
	DEF	ERRORX	THEN	314	0,44
	DEF	TAB.	TAB	315	20,45
	DEF	STEP.	STEP	316	0,41
	DEF	EXOR.	EXOR	317	2,51
	DEF	NOT.	NOT	320	7,50

```
DEF INTDIV          DIV (\)      321   12,51
DEF ERNUM.          ERRN         322   0,55
DEF ERRL.           ERRL         323   0,55
DEF RESET.          RESET        324   0,44
DEF AND.            AND          325   4,51
DEF MOD10           MOD          326   12,51
DEF ERRORX          ELSE         327   0,44
DEF SIN10           SIN          330   20,55
DEF COS10           COS          331   20,55
DEF TAN10           TAN          332   20,55
DEF NOP2.           TO (ASSIGN) 333    77,51
DEF RSTO..          RESTORE LN   334   0,227
DEF ERRORX          DUMMY        335   0,44
DEF ERRORX          [            336   0,44
DEF ERRORX          ]            337   0,44
DEF INTDIV          \            340   12,51
DEF POS.            POS          341   52,55
DEF DEG10           RTD          342   20,55
DEF RAD10           DTR          343   20,55
DEF INT5            FLOOR        344   20,55
DEF ERRORX          DUMMY        345   0,44
DEF READN.          READ (NUM)   346   0,44
DEF ULIN#.          USING LINE #347    0,327
DEF INPUN.          INP NUMERIC 350    0,33
DEF INPU$.          INP STRING   351   0,33
DEF FNRET.          LET FN(;;=) 352    0,16
DEF READS.          READ$        353   0,44
DEF PRLINE          PRINT END    354   0,35
DEF SEMIC.          PRINT;       355   0,36
DEF COMMA.          PRINT,       356   0,36
DEF SEMIC$          PRINT;$      357   0,36
DEF COMMA$          PRINT,$      360   0,36
DEF ERRORX          DUMMY        361   0,241
DEF STEPK.          STEP KEY     362   0,241
DEF FTADR           1 DIM ARRAY 363    0,1
DEF FTADR           2 DIM ARRAY 364    0,1
DEF TEST.           TEST KEY     365   0,341
DEF ERRORX          DUMMY        366   0,44
DEF ERRORX          DUMMY        367   0,44
DEF ROM:GO          EXTERNAL ROM370    0,214
DEF BP:GO           BINARY PROG 371    0,214
DEF ERRORX          DUMMY        372   0,44
DEF ERRORX          DUMMY        373   0,44
DEF ERRORX          DUMMY        374   0,44
DEF ERRORX          DUMMY        375   0,44
DEF ERRORX          DUMMY        376   0,44
DEF ERRORX          DUMMY        377   0,44
```

NOTES

Appendix G

ERROR MESSAGES

Below is a list of the error messages provided by the HP-83/85 Assembler ROM and the System Monitor. For other errors refer to the HP-83 or HP-85 Owner's Manual or to the manuals for other peripheral devices that may be attached to the computer.

ASSEMBLER SYSTEM ERRORS

Error Message	Error Condition
ERROR 109: ILL MODE	A command has been executed in the wrong operating mode (e.g., ASSEMBLER has been typed when computer is already in assembler mode).
ERROR 110: LBL	An invalid label has been seen; may have been either longer than six characters or beginning with a digit.
ERORR 111: OPCO	The opcode is not recognized; may have been because of misspelling, because there was no space between a label and the opcode, or because the opcode was entered in the first or second column after the line number.
ERROR 112: ARP-DRP	Invalid ARP or DRP; ARPs and DRPs must be between Ø and 77 inclusive, and cannot be 1.
ERROR 113: OPER	Bad operand; e.g., LDM R34, = 3, remark. Because a number follows the equal sign in this example, the assembler expects another number after the comma. Also, each literal value must be specified with two digits if a BCD quantity.
ERROR 114: FIN-LNK	Missing FIN or LNK statement. If the file name or file type is wrong in the LNK statement, then a "FILE NAME" or "FILE TYPE" error will be generated.

Error Message	Error Condition
ERROR 115: ASSM ROM	At power-on, this means the ROM had a checksum error. At a breakpoint, all errors generate this message.

ASSEMBLY-TIME ERRORS

Error Message	Error Condition
ILL NAM	A NAM statement has already been executed, or an ABS ROM has been executed.
AIF UND	The specified conditional assembly flag has not yet been defined as set or cleared.
ILL ABS	An ABS or NAM statement has already been encountered.
JMP FROM	The jump from that line is out of range.
JMP TO	The jump to that line is out of range.
UND LAB	After assembly was completed, this label had not been defined either in the program or in the optional global file.
ILL GLO	The GLO statement occurs after a NAM statement, ABS statement, or another GLO statement.

APPENDIX H

PROGRAMMING HINTS AND ADDENDA

1. If execution of certain Advanced Programming ROM statements is attempted in assembler mode, unpredictable results can occur. These AP ROM statements are:

X REF L
X REF V
SCAN
REPLACE VAR.

NOTES

INDEX

T

HEWLETT
PACKARD

For additional information please contact the nearest authorized Series 80 HP-83/85 dealer or your local Hewlett-Packard sales office.

00085-90444 Rev. B 11/81

Printed in U.S.A.